UNIVERSITEIT • STELLENBOSCH • UNIVERSITY

Video Camera Design and Implementation for Telemedicine Application

by

Kibreab Ghebrehiwet Behaimanot

Thesis presented at the University of Stellenbosch in partial fulfilment of the requirements for the degree of

Masters of Science

Department of Electrical and Electronic Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Study leader: Dr. Mike Blanckenberg

November 2004

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: .....
K.G. Behaimanot

Date: .....

Abstract

Video Camera Design and Implementation for Telemedicine Application

K.G. Behaimanot

Department of Electrical and Electronic Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Thesis: MScEng

November 2004

Primary health care telemedicine services require the acquisition and transmission of patient data including high quality still and video images via telecommunication networks.

The objective of this thesis is to investigate the implementation of a general-purpose medical camera as an alternative to the complex and costly CCD based cameras generally in use at present. The design is based on FillFactory's SXGA (1280 $\times$1024) CMOS image sensor.

A low-cost Altera Cyclone FPGA is used for signal interfacing, filtering and colour processing to enhance image quality.

A Cypress USB 2.0 interface chip is employed to isochronously transfer video data up to a maximum rate of 23.04 MBytes per second to the PC.

A detailed design and video image results are presented and discussed; however the camera will need repackaging and an approval for medical application by medical specialists and concerned bodies before releasing it as full-fledged product.

Uittreksel

Video Camera Design and Implementation for Telemedicine Application

K.G. Behaimanot

Department of Electrical and Electronic Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Tesis: MScEng

November 2004

Primêre gesondheidssorg telemedisyne dienste moet hoëkwaliteit televisiebeelde van hul pasiënte verkry deur van telekommunikasienetwerke gebruik te maak.

Die doel van hierdie tesis is om die toepassing van n meerdoelige mediese kamera te ondersoek as n alternatief tot duur, komplekse CCD-gebaseerde kameras wat huidiglik gebruik word. Die ontwerp is gebaseer op n hoëkwaliteit CMOS beeldsensor.

n Goedkoop Altera Cyclone FPGA word gebruik vir seinkoppelvlak, filtering en kleurprosessering om die kwaliteit van die beeld te verhoog.

n Hoëspoed USB 2.0 poort word gebruik om die data teen die nodige spoed te versend.

n Gedetailleerde ontwerp, en die beeldresultate word voorgelê en bespreek. Die kamera moet egter eers deur mediese spesialiste en relevante beheerliggame goedgekeur word voordat dit as n volledige produk vrygestel kan word.

Acknowledgements

I would like to thank my study leader, Dr. Mike Blanckenberg, for his continuous guidance, insight and patience. His accommodating behaviour was behind the motivation that enabled me to keep pushing through every problem I have encountered.

The success of this project would have not been possible without the help of Electronic Systems Lab (ESL) members, who were always ready to share their mind and time.

I am indebted to all Eritrean friends specially to my flat-mates Esayas Welday and Yared Tesfay for their continuous encouragement and help.

I would like to thank the Government of the State of Eritrea and Dr. Mike for providing the required funding.

My deepest thanks is to my beloved family for their precious love, care and encouragement.

Praise be to God, my Father in heaven, for giving me the strength, health and above all for His mercy and love in my life.

Dedications

To the glory of God

Contents

List of Figures

List of Tables

List of Abbreviations

CCD	Charge Coupled Device
CIF	Common Interchange/Intermediate Format
CMOS	Complementary Metal Oxide Semiconductors
CT	Computed Tomography
DDK	Driver Development Kit
DSP	Digital Signal Processor
FIFO	First-In First-Out
FPGA	Field Programmable Gate-Array
FPS	Frames per Second
I/O	Input and Output
I2C	Inter-IC Control
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
ISDN	Integrated Service Digital Network
JPEG	Joint Photographic Experts Group
JTAG	Joint Test Action Group
Kb	Kilo bits
KB	Kilo bytes
Kbps	Kilo bits per second
KBps	Kilo bytes per second
LAN	Local Area Network
LED	Light Emitting Diode
LSB	Least Significant Bit

MB	Mega bytes
Mbps	Mega bits per second
MBps	Mega bytes per second
MOSFET	Metal Oxide Field Effect Transistors
MPEG	Moving Picture Experts Group
MRI	Magnetic Resonance Imaging
MSB	Most Significant Bit
OHCI	Open Host Controller Interface
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PLL	Phase-Locked Loop
RAM	Random-Access Memory
RGB	Red-Green-Blue
ROM	Read-Only Memory
SCIF	Sub Common Interchange/Intermediate Format
SDK	Software Development Kit
SRAM	Static Random Access Memory
SRB	Stream Request Block
SVGA	Super Video Graphics Array
SXGA	Super Extended Graphics Array
TTL	Transistor-Transistor Logic
USB	Universal Serial Bus
VGA	Video Graphics Array
VHDL	VHSIC Hardware Description Language
XGA	Extended Graphics Array
YUV	Luminance-Bandwidth-Chrominance

Chapter 1

Introduction

1.1 Telemedicine Defined

In general Telemedicine can be defined as health care services delivered via telecommunications networks. Telemedicine practices may be as simple as normal voice communication between two medical experts over telephone line. Whereas a complex form of telemedicine could include transmission of complex patient data, multi-point live video conferencing and remote robotic surgery.

The most widespread form of telemedicine is Telediagnosis, where a doctor makes a diagnosis based on data transmitted from a remote location [1]. The data can be simple stethoscopic sound, digital blood pressure reading, X-rays, MRI CT scans, or real-time video.

In this way telemedicine enables patients in remote areas to get easy and quick access to medical services regardless of their geographical location. Moreover, it helps medical professionals to stay close to larger medical centres, where they can enjoy better facilities, good living conditions and have better opportunity to update their professional skills.

In addition the use of telemedicine not only breaks the geographical barrier but also cuts down the overall cost of delivering medical service to a great extent. In future, with the advancement in computer and telecommunication technology, cost of telemedicine based health care services are expected to drop significantly. However, a number of technical problems and limitations are still to be addressed in today's telemedicine. The next section discusses those challenges putting more

emphasis on video related issues.

1.2 Major Challenges in Telemedicine

Telemedicine is built up on a number of technologies including computers, communication networks, video, and specialized medical equipment. This fact makes telemedicine design projects to involve team work, where each group or individual works on a specific part of the project. Therefore it is difficult to discuss all problems and associated limitations of each technology, nevertheless it suffices to give a brief discussion of the major problems and their implication to the overall quality requirement telemedicine entails. The following sub-section discusses the bandwidth problem in telemedicine.

1.2.1 Bandwidth constraint

Telemedicine is basically concerned with the transmission of medical data between two places, be it within the same hospital or across a continent. If the physical distance in consideration is short (in the order of meters), Local Area Network (LAN) is the best solution. Its main advantages are the wide bandwidth it offers and the low running cost required once it is installed.

The most commonly used communication channel for data transmission between rural and urban areas is the Integrated Service Digital Network (ISDN), with "basic rate" of 56kbps or 64 kbps. This data rate is very slow for transmission of uncompressed video data. To illustrate this, consider the transmission of uncompressed 5-sec video clip of SCIF (320 x 240) resolution, 24-bit true colour at 15 fps. The total size of the video clip will be:

$$File\ Size = 320 \times 240 \times 24bits \times 15frames/sec. \times 5sec.$$

$$File\ Size = 138.24Mb = 17.28MB$$

Transmitting this video clip via ISDN (64kbps) will take more than 30 minutes. Besides the big telephone line cost, it is totally unacceptable for a busy physician to spend a significant amount of time simply waiting for the data to arrive. This transmission bottleneck can be overcome by applying image compression techniques, which are briefly discussed in the next section.

1.2.2 Video compression and image quality considerations

In general there are two types of image compression: lossless and lossy. Compression of medical images should preferably be lossless; however the compression ratios are limited to between 2:1 and 4:1 [1]. On the other hand lossy compression methods can yield compression ratios between 10:1 and 20:1 [1]. In medical imaging however they have been traditionally viewed with caution as they permanently lose some of the data, which are thought of as redundant in the compression method.

The need for compression and the demand for high quality images in medical application seem to contradict each other. However, recent research has shown that compression methods such as JPEG and wavelet-based compression can give diagnostically lossless images with compression ratio exceeding 10:1 [4]. Of course it is not easy to get general rules or guidelines that work in all medical image types, but there is some evidence that lossy compression methods in telemedicine may not adversely affect diagnostic accuracy of certain medical images. A study on angiography showed that JPEG lossy compression of 6-14:1 does not affect the diagnosis significantly. It also mentions that a compression ratio of up to 30:1 might be used for telemedicine applications [4]. Other studies also have shown that lossy compression methods in telemedicine may not adversely alter the accuracy of images in remote consultation for dermatology [5] and ultrasonography [4].

Most commercial digital video cameras use some sort of compression in order to overcome the bandwidth bottleneck in PC interface. The majority of the compression methods are based on lossy algorithms such as Motion-JPG and MPEG. The resulting video stream after a compression/decompression operation with lossy algorithms is of lower quality due to data loss during compression and adds unwanted artifacts during decompression. Degradation in quality is tolerable and not apparent to the viewers in applications such as video entertainment; however image quality in cameras for medical application can not be compromised.

Evaluating the different compression algorithms and/or investigating their implementation requires a full study on its own. Therefore, it was decided that the scope of this design project would be limited to capturing uncompressed video. Once uncompressed video is successfully captured, the already developed codec

softwares such as H263 (commonly used for "basic rate" ISDN) could be used for compression/decompression purposes.

More broader discussion on the scope and objectives of the thesis is given in the next section. This section also gives the general report outline and how the report is organized.

1.3 Thesis background and its scope

As part of ongoing telemedicine project work, aimed at developing a low-cost telemedicine workstation for use in the African continent, this thesis was first intended to come up with a low cost video module system design. The study involved selecting a proper camera module, choosing a proper PC interfacing method, and possibly software image compression for later transmission via ISDN line. While exploring the available camera modules on the market, it was found that cameras, especially those for medical application are very expensive.

The need to keep the total cost as low as possible and the requirement to capture uncompressed video (as explained in the previous sub-section) led to a proposal to investigate the development of an affordable general-purpose medical camera, which can capture real-time uncompressed video.

Additional motivation to support the proposal was the fact that designing a digital video camera from scratch was more challenging and offers the opportunity to acquire a good knowledge of all the stages involved in developing a peripheral PC device. Besides being an ideal ground for working on hardware, firmware and software design, the study could be used for further academic research on digital imaging.

The report is organized in seven chapters. The first chapter being an introduction (current chapter), chapter 2 gives an overview of the whole design process in block diagram and explains each block in fair detail. The hardware selection process and criteria used in the selection process are given in chapter 3. A summary of the data sheet of major IC components is given in appendix A. All hardware design details including schematic and PCB design, VHDL and firmware design are presented in chapter 4. Chapter 5 discusses main software design issues including operating system choice, USB camera driver design and an application

program development. A fairly detailed driver and application program design is also presented in the chapter. Chapter 6 covers the various hardware and software performance results. Although the chapter presents video and still image results, no image quality evaluation (qualitative and/or quantitative) is done. Finally the conclusions and recommendations are summarized in Chapter 7.

Even though some information, relevant to the material in the report, is included in appendices A to E in print, all codes and accompanying design information are kept on CD (appendix F) to keep the report volume to a reasonable size.

Chapter 2

Design Flow Choices

2.1 The Big Picture

Even though design details of a PC based video camera varies from one design to the other, in most cases the overall design flow assumes similar layout. Figure 2.1 shows typical block diagram representation of a PC video camera.

Figure 2.1: Typical block diagram design of a PC video camera

The first block represents a solid-state image sensor. The die-sized silicon chip contains millions of photosensitive diodes called photosites. Each photosite reacts to the light that falls on it by accumulating charge; the more light, the higher the charge accumulated. The accumulated charge is then converted to voltage signal and finally it is digitized and is output to the external pins of the image sensor.

An image sensor is incapable of capturing an image on its own. It requires external circuitry to provide necessary timing and control signals. Most colour image sensors output unprocessed raw video, therefore external processing is required to convert the raw video to a standard format video. The image acquisition and processing block represents these functions and any other image processing operation.

Once the raw video is converted to a standard format and is further processed to enhance image quality, the next task is video data interfacing to the PC. To

obtain a smooth data transfer rate to the PC, the video data can be temporarily stored in memory. The PC interface controller controls the data transfer to and from the PC. It may also take part in actively responding to PC commands and reconfiguring the camera hardware.

The final stage involves a PC to display the incoming video stream. Interfacing a peripheral device to a PC requires knowledge of the device driver and operating system. The camera driver handles all communication details between the camera hardware and the operating system. If an appropriate driver can not be found among the operating system drivers, the designer is forced to write a device driver that handles all peripheral device communication details. In addition an application program, which displays the video and provide user friendly camera control is required.

The following sections (2.2 and 2.3) give a fuller description of video image processing and PC interface tasks. To keep the chapter volume small, the discussion on software issues including operating system, device driver and application program is deferred to chapter 5.

2.2 Colour Image Processing

Image processing is a very vast subject. Due to the limited scope of this thesis, the report will present only a brief discussion of the image processing tasks. More emphasis will be given to image processing techniques relevant to the work presented in this thesis.

By and large image processing can be broken in to a chain of processes. Figure 2.2 shows a block diagram representation of a typical image processing chain.

Figure 2.2: General image processing chain

Before discussing the colour image processing blocks in the figure above, explanation on how an image sensor captures colour image is given in the next section.

2.2.1 Colour image capture

Photosites in an image sensor record the total intensity of light that falls upon them and hence they do not differentiate between colours. However there are different systematic ways of enabling an image sensor to capture colour image.

The most basic way uses a prism to split the incoming light into its primary colours (red, green and blue) and then the split light rays are cast onto three separate image sensors. The output from the three sensors makes up the red, green and blue components of the captured image. Cameras using this technique (commonly called 3CCD cameras) guarantee the highest quality. However due to the need for using three sensor chips, a highly accurate mechanical design and the special lenses they employ, their price is very high.

To bring the cost down, most digital colour cameras use a single image detector covered by a three-colour mosaic known as Colour Filter Array (CFA). The most popular CFA pattern used in image sensors is the Bayer RGB colour filter pattern shown in figure 2.3.
It alternates rows of red and green filters with rows of blue and green filters. As a result, the number of green colour filters is twice as much as red or blue filters. More green filters were used to take advantage of human eye perception. The human eye is more sensitive to green light (which is significantly perceived as luminance) than it is to red or blue (perceived more as chrominance signals). Hence, the Bayer RGB pattern provides high spatial frequency in luminance (green) at the expense of chrominance signals (red and blue).

Figure 2.3: Bayer RGB CFA

The CFA allows only one colour to pass and illuminate a photosite. The recovery of full-colour images from a CFA-based detector requires some sort of interpolation to fill the missing values at each pixel place. These methods are commonly referred to as colour demosaicking algorithms. Some of the methods are discussed shortly in the following section.

A new breakthrough in colour image sensors has been recently announced by Foveon inc. Unlike the traditional mosiac-filter image sensors, the Foveon image sensor captures red, green and blue colour information at every pixel point. The core idea behind the technology is the fact that light with different wavelength is absorbed at different depths in silicon[17]. As shown in figure 2.4, the red, green and blue photodetectors in the image sensor are located at different depths. The blue ones are located near the surface of the sensor, the green ones in the middle, and the red ones in the bottom of the sensor.

Figure 2.4: Foveon image sensor - layered photodetectors [17]

The image sensor captures sharper image and has better colour reproduction. Because the captured image is already a full-colour RGB, the need for demosaicking operation is eliminated.

However in this design the Foveon Image sensor was not considered for selection mainly because of unavailability in the market at the time of image sensor selection. In addition the idea of designing a 3-image sensor camera was dropped mainly to avoid the complex mechanical and optical design.
Therefore for this design only a single sensor camera with Bayer RGB CFA was considered. The next section discusses demosaicking operations for Bayer RGB colour image sensors.

2.2.2 Demosaicking

The demosaicking process involves populating the missing pixel values at each pixel point by an estimated value from the surrounding pixels. Figure 2.5 shows the demosaicking process applied to the sub-sampled red, green, and blue planes. The demosaicked plane consists of a complete three colour plane representing a full-colour image.

Figure 2.5: Bayer RGB demosaicking operation

Even though this thesis considered a number of demosaicking methods, in this report a very brief description of a few interpolation methods is given. Finally a suitable method is selected based on a complete thesis paper on demosaicking [16]. The criteria for selection were best output image quality and less computational intensiveness.

Nearest neighbour interpolation

Nearest neighbour interpolation is the simplest demosaicking method. In this method, the missing pixel is assigned the value of the nearest pixel in the neighbourhood. If there are more than one pixel values that can equally be candidates to fill the missing pixel, one of these is chosen.

Although nearest neighbour interpolation is simple and computationally less intensive, quality of the demosaicked image is poor.

Bilinear interpolation

The bilinear interpolation method uses an average of all the sub-sampled pixel values in the neighbourhood to evaluate the value of the missing pixel. Consider figure 2.6.

Figure 2.6: Bilinear interpolation of Bayer RGB CFA

At any red centre the four neighbouring green and blue sub-samplers are averaged to give an estimate of the green and blue values. For example at R_{33}:

$$G_{33} = \frac{G_{23} + G_{32} + G_{34} + G_{43}}{4} \tag{2.1}$$

$$B_{33} = \frac{B_{22} + B_{24} + B_{42} + B_{44}}{4} \tag{2.2}$$

At green centres the neighbouring two red and blue sub-samples are averaged. Consider G_{34} :

$$R_{34} = \frac{R_{33} + R_{35}}{2} \tag{2.3}$$

$$B_{34} = \frac{B_{24} + B_{44}}{2} \tag{2.4}$$

Similarly at blue centres the neighbouring four red and green sub-samples are averaged to populate missing red and green values. For instance at B_{44}:

$$G_{44} = \frac{G_{34} + G_{43} + G_{45} + G_{54}}{4} \tag{2.5}$$

$$R_{44} = \frac{R_{33} + R_{35} + R_{53} + R_{55}}{4} \tag{2.6}$$

Freeman algorithm (Median Filter)

The Freeman algorithm consists of two steps; a Bilinear interpolation followed by median filtering of colour difference. Considering the fact that the Bayer RGB pattern in figure 2.3 samples green more at the expense of red and blue, it is logical that the demosaicking process will result in more accurate green values than for red or blue. Freeman algorithm tries to correct interpolated R and B planes using the more accurate green image plane[18]. The green plane is left untouched when doing the median filtering. The qualitative description of the algorithm is as follows: First two difference image planes $R - G$ and $B - G$ are created. Next these colour differences are median filtered and finally R and B output planes are created by adding the G plane to the median filtered $R - G$ and $B - G$ planes. Finally the R and B output planes are used to replace the bilinearly estimated R and B pixel values.

Choice of demosaicking algorithm

Comparing output image quality of various demosaicking algorithms requires series of tests using different test images. Additional research may also be required to relate the results with human vision. However in this design the results of a full thesis paper on demosaicking [16] was consulted. The experimental study observed that for the majority of test images considered, Freeman algorithm

Figure 2.7: Spectral response of an image sensor

gives the smallest error measure. However it performs poorly when the image consists of sharp edges like in vertical and horizontal stripe images.

In medical images sharp edge vertical and/or horizontal stripes are very uncommon; rather the images are smooth and gradually changing. Therefore for this design, the Freeman algorithm was selected as demosaicking method.

Another criterion was the computational intensiveness of the algorithms. As it shall be pointed out later in this chapter, the Freeman algorithm can be easily implemented in hardware as it is computationally less intensive compared to other rival algorithms.

2.2.3 Colour processing

The colour processing involves operations such as colour correction, colour space conversion, edge enhancement, and gamma correction. To keep the report short, only a brief discussion on colour correction (implemented in this design) is given.

Colour correction

For different lighting types and conditions, colour cameras give image output of different colour composition . In order to reduce or eliminate this undesired effect, colour correction, commonly referred as white balance, is required. The aim of white balance is to give the camera a reference to white. White balance is done by exposing the image sensor to a standard white and the camera will record the colour temperature and use this to render subsequent images correctly.

Colour rendition problem may also arise due to non-linearities in the spectral response of the image sensor. Figure 2.7 shows a spectral response of an image sensor manufactured by Fillfatory.

Colour correction can be performed by multiplying the RGB pixel values of the image by a 3x3 matrix as shown below.

$$\begin{bmatrix} R_c \\ G_c \\ B_c \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.7}$$

Generating the correct values of the 3x3 colour matrix requires a great deal of knowledge in image science. However, in this design standard colour palette (ColorChecker) was employed to estimate the matrix. Following are the steps used in the calculation:

- The R, G, and B values of the colours in the original colour palette were first put in to a matrix.

$$O = \begin{bmatrix} R_1 & R_2 \dots B_{24} \\ G_1 & G_2 \dots B_{24} \\ B_1 & B_2 \dots B_{24} \end{bmatrix} \tag{2.8}$$

- A photo of the standard colour palette was taken and samples were entered to a matrix

$$C = \begin{bmatrix} R_1 & R_2 \dots B_{24} \\ G_1 & G_2 \dots B_{24} \\ B_1 & B_2 \dots B_{24} \end{bmatrix} \tag{2.9}$$

- Then an optimal linear transformation matrix A that best maps the colour samples of the photo taken by the camera C into the corresponding original colour samples O was calculated using least square method.

$$O = A * C \tag{2.10}$$

$$A = O * C^{-1} \tag{2.11}$$

- If we have more than nine independent RGB samples (more than the number of unknowns in the matrix A), then the set of linear equations becomes over-determined and the least-squares solution can be used.

2.2.4 Compression

Image Compression techniques reduce image data size by discarding redundant information contained in the captured image. A large number of image compression algorithms and standards are in use today. The most common compression standards include Joint Photographic Experts Group (JPEG) standard for still image and moving Picture Experts Group (MPEG) video compression standard.

Video compression algorithms can be divided into those which use inter-frame and those employing intra-frame compression algorithms. With intra-frame algorithms such as M-JPEG, each picture frame is compressed independently, whereas in inter-frame, compression is performed both with in same picture frame and in between adjacent picture frames. Several video compression standards, tailored for specific applications, are available. For instance H.263 is a well-known video conferencing standard.

A receiver of a compressed video requires a reverse processes of decompression before it can display the image. Compression/decompression algorithms and standards continue to change along with the rapidly changing technology to meet the rising demand for efficient and high quality image compression. Recently released JPEG2000 standard can be used for wide range of applications starting from commercial digital cameras to advanced medical imaging. Unlike the well-known MPEG family of standards, Motion JPEG 2000 (part of JPEG2000) can be used to compress video losslessly.

As it was discussed in the introduction part, even though video compression will ultimately be required in real telemedicine applications that use narrow bandwidth communication networks, in this design no compression was considered. However the compression can be done later in software using standard codecs available as part of the PC operating system.

The main video processing algorithms implemented in this design are:

- Bilinear colour interpolation (Demosaicking)
- Median image filtering to increase image sharpness and remove noise
- Colour correction

2.2.5 Implementation of video processing algorithms

There are several hardware and software alternatives for implementing image processing algorithms. These alternatives ways give varying level of performance benefits. However, these benefits should be compared against other factors including cost and design time. This section presents some implementation options.

Software implementation

The software option, where a PC is used to perform video processing operations on the incoming video stream, is relatively simple and requires less development time. Moreover debugging programming errors and updating in already developed software packages is easy. Furthermore a number of image processing softwares are available. However, with the demand for high resolution video (which results in very high data rate), PC based video processing solutions are becoming less useful mainly because they become too slow to handle the fast streaming video data.

For this reason, this design resorted to use a dedicated hardware that does all video processing operations requirements. Hardware implementation of video processing offers the advantage of increased processing speed lessening the burden on the PC. However, it requires a prolonged development time. Moreover debugging and/or design updating will be a lengthy processes. The following sub-section discusses hardware video processing options.

Dedicated hardware option

The most common hardware implementation of Image processing algorithms includes: Digital Signal Processor (DSP), reduced instruction cycle microprocessor (RISC processor) and a Field Programmable Gate Array (FPGA). Traditional DSP processor architectures have only been capable of performing low complexity image processing functions in real time and this is still the case even with the latest DSP processors from Texas Instruments and Analog Devices. Similarly RISC microprocessors are too slow to perform real-time video processing operations mainly due to sequential instruction execution which takes a number of clock cycles per instruction.

Field Programmable Logic Array (FPGA) implementation of video processing algorithms is becoming increasingly popular as modern high-density FPGAs incorporate greater functionality including embedded memory, PLL clock management, embedded processor, and DSP blocks.

Growing capabilities of these FPGAs has enabled handling of more heavy processing requirements, which were normally performed using DSP. These advanced FPGAs have proven themselves more than capable to handle advanced, computationally-intensive algorithms and applications. Altera, an FPGA manufacturer, produces a family of FPGA solutions to meet different design needs. For instance the newest product family Stratix, offers up to 28 DSP block and up to 10Mbits of memory and up to 12 PLL blocks on a single chip. The rich resource in such FPGA is an ideal solution for video processing implementation tasks. Therefore, an FPGA can be regarded as good replacement for the traditional DSP and microprocessor based solutions.

In this design an FPGA implementation was selected for the following reasons:

1. Single chip solution for virtually all image processing needs;

2. Availability of programmable I/O ports which provides convenient signal interface with image sensor as well as pc interface controller.

3. Parallel algorithm implementation, which increases the processing speed.

4. Market availability

2.3 PC Interface Control

Data communication between a PC and a peripheral device can be established using many possible PC interface technologies. In general PC data communication methods can be grouped as parallel and serial communication methods.

Table 2.1: Comparison of popular computer interfaces.

Over time several parallel and serial signalling standards have been defined for data communication interfaces. The most common standards include RS232 (PC serial port), RS422, RS485, Universal Serial Bus (USB) and IEEE1394 (Firewire). RS232 is becoming obsolete and is being replaced by the more robust and easy to use USB. Table 2.1 shows a comparison between different PC communication standards with regard to maximum data transfer rate, maximum cable length and the number of devices that can be attached to a the PC.

In general video devices require very high speed data interfaces. Consider a video camera with uncompressed RGB24 video format and capturing a VGA resolution image at 25frames/sec. The video output data rate will be:

$$Data\,Rate = 640 \times 480 \times 24bits \times 25fps$$

$$Data\,Rate = 23.04MB/sec.$$

This large amount of data can only be handled by either USB or IEEE1394 (Firewire). In this design USB2.0 and IEEE1394 were considered for video data interface to a PC.

IEEE1394 better known as firewire (a name introduced by Apple) and USB 2.0 (USB high speed) are the competitive modern buses for PC peripheral communication. These two bus technologies are better and faster than all of the classic bus types such as RS-232, Parallel Port/ECP/EPP and even local buses like ISA.

USB is a shared serial bus and was introduced to replace most of the traditional PC ports. It is a versatile and user-friendly interface. The first USB specification (USB 1.0) was released in 1996 and later in 1998 USB 1.1 was released. USB1.1 fixed USB 1.0's problems and introduced new transfer type (Interrupt OUT). USB 1.0 / 1.1 (USB1.x) has a bandwidth of 12Mbits per second and it is limited to low speed devices (such as keyboard and mice) and full speed devices (such as printers and webcams). Compared to IEEE1394 (400Mbits per second speed), USB 1.x is extremely slow. However the introduction of USB 2.0 specification has dramatically increased the bandwidth to 480Mbits per second, which made USB faster than IEEE1394. But with the introduction of the proposed IEEE1394b, Firewire bandwidth will increase up to 3.2Gbits per second. IEEE1394, once again will take the lead [19].

USB places most of the interface intelligence inside the host computer, therefore enables the design of less complex and inexpensive peripherals. Unlike USB, IEEE1394 architecture is not host centric. Its peripherals do have the intelligence not only to initiate communication with the PC but also to establish direct peripheral to peripheral communication. The interface is really flexible, but the peripheral electronics employed is much more complex and expensive compared to USB. Table 2.2 gives a comparison of USB and IEEE1394.

Table 2.2: USB 2.0 Vs IEEE 1394 comparison

USB has become the standard peripheral interface in the PC world. Nowadays most newly purchased PC motherboards have a built-in USB 2.0 host controller.

In this design USB 2.0 was preferred to IEEE1394 mainly:

- To keep costs down;
- Reduce design complexity;
- Take advantage of availability of built-in USB 2.0 port in newly purchased PCs.

More discussion on USB is given in appendix D. It discusses topics relevant to the design work in this thesis: including USB peripheral device topology, enumeration process, USB transaction and USB transfer types.

USB supports four transfer types : Control transfer, bulk transfer, interrupt transfer, and isochronous transfer. In the design isochronous transfer protocol was selected mainly to take advantage of guaranteed bandwidth, thereby ensuring constant data arrival.

Software design for PC video camera or any other peripheral device requires careful thought of the operating system to be used, device driver requirements and GUI application program development environment. All software issues are discussed in chapter 4.

Chapter 3

Hardware Selection

3.1 Image Sensor

3.1.1 Introduction

Since their discovery in the seventies charge coupled device (CCD) sensors have developed towards a mature product through their use in many imaging applications such as astronomical telescopes, scanners, and video camcorders. Because of their superior performance and popularity the sensors had the control of all imaging market needs. However, in the last few years, strong efforts made to improve CMOS (Complementary Metal-Oxide Semiconductor) sensor performance enabled them to be competitive to the long-established CCDs.

There have been speculations by advocators of CMOS that the continuous advancement in CMOS sensors will eventually bring CCDs to an end. On their part CCD supporters have been presenting the superior performance of CCD as unbeatable. This report will make general comparison between these competitive technologies based on key variables including performance, level of integration and cost. Towards the end of this section, the focus will turn into selection of an appropriate image sensor for medical imaging application.

3.1.2 CCD and CMOS architecture comparison

Even though both CCD and CMOS sensors are based on silicon material, their architectural design and manufacturing process differ considerably. As it is depicted in figure 3.1, pixels (light sensing elements) in a CCD sensor are arranged in an X-Y matrix of rows and columns. Each pixel contains a photodiode and an adjacent charge holding region (potential barrier and well) [6]. The photodiodes intercept incoming light photons and convert them into charge (electrons) through photoelectric effect while the potential barrier structure keeps the electrons from leaking. The amount of charges collected is proportional to the illumination intensity and exposure time.

Figure 3.1: Inter-line CCD Architecture and associated external circuitry [7, 8]

Once charge has been integrated and held locally by the bounds of pixel architecture, the packets of charge are shifted sequentially to common output structure where electron-to-voltage conversion takes place [8]. Depending on the method of shifting charge packets CCDs are divided into interline-transfer, frame-transfer and frame-interline-transfer CCDs [7]. Figure 3.1 shows an inter-line CCD, where the unshaded squares represent the photodiodes and the adjacent shaded blocks represent vertical CCD registers. After the exposure time is elapsed, the charge packets are shifted from the pixels to the shift registers. Then the vertical shift registers are emptied in to the horizontal-output register, one horizontal line at a time. Finally these charge packets are converted into voltage and amplified before sending the analog signal off the chip for further processing.

Figure 3.2: (a) Passive and (b) Active pixel CMOS architecture [9]

Similar to the CCD, CMOS pixels are arranged in X-Y matrix fashion, and each pixel contains a photodiode to convert light to electrons. However the CMOS sensor, on top of the photodiode, contains a charge-to-voltage converter, a reset

and select transistor, and an amplifier at pixel level. This architecture makes it possible to readily integrate peripheral electronics devices like timing circuits, analog to digital converter and digital logic. The architecture also allows signals to be read from the entire array, a portion of the array, or a single pixel in the array by simple X-Y addressing.
CMOS image sensors come in two architectures, passive and active. These are shown in figure 3.2. Unlike passive CMOS sensors, in active pixel sensors amplification is done at pixel level, causing active pixel sensors to have increased dynamic range and reduced SNR.

3.1.3 CCD Vs CMOS performance comparison

As a result of architectural and manufacturing process differences, CCD and CMOS image sensors have considerable performance difference. These differences are briefly discussed in this sub-section.

Dynamic range

Dynamic range indicates the ratio of the maximum signal output (pixel saturation level) to its dark noise level [6]. CCD sensors have some advantage in dynamic range because of less on-chip circuitry, inherent tolerance to bus capacitance variations and common output amplifiers with transistor geometries that minimize noise [10].

Signal to noise ratio (SNR)

The definition of SNR is similar to dynamic range except now the total noise is considered instead of only the dark noise level.

Dark current

Dark current is used to refer to a background signal present in the image sensor readout when no light is incident upon the image sensor. Dark current is a result of thermally emitted charge being collected in the photosites [6].

CMOS sensors have dark current ranging from 100 to 2000 pA/cm2 while CCD manufacturers reduce dark current to a level as low as 10 pA/cm2 [10].

Fixed pattern noise

If the output of an image sensor under no illumination is viewed at high gain a distinct non-uniform pattern, or fixed pattern noise, can be seen. Dark fixed pattern noise is mainly caused by small variation in "pixel geometry" at manufacturing time and both CMOS and CCD image sensors have a comparable fixed pattern noise level [10].

Fill factor

Fill factor refers to the percentage of a light sensitive area to the total photosite area. CCDs have a 100% fill factor [10] but CMOS cameras due to an optically dead space occupied by the metal-oxide-field-effect-transistors (MOSFETs) have much less fill factor. As a result of this CCD sensors are preferred in applications such as telescopes, and satellites where illumination level is very low.

Level of integration and power consumption

CMOS sensors offer superior integration capabilities. They can incorporate other circuits such as the clock drivers, timing logic, and signal processing on the same chip, eliminating the many separate chips required for a CCD. This makes the camera smaller, lighter, and cheaper. It is technically easier to design a CMOS camera which integrates many functions on-chip than to design a CCD camera which requires accurate timing and analog processing circuits. Besides CMOS sensors consume very low power and require a single power supply voltage. This makes them ideal for PC interface via USB, where the PC can provide the required power at single voltage level (5V). CCDs on the other hand dissipate high power (as much as 5 times that of CMOS sensors) and require multiple non-standard supply voltages (such as 15V, 20V DC).

Cost

The fact that cost of fabricating a CMOS wafer is lower than the cost of fabricating a similar wafer using the more specialized CCD process, makes CMOS sensors to be relatively cheap. If the cost of the on-chip circuit functions such as

timing generation, biasing, analog signal processing and digitizing are considered the overall system cost becomes much cheaper.

The advantages of using CMOS sensors can be summarized as:

1. Fully Integrated solution offering all digital I/O

2. Single voltage supply, and reduced power consumption.

3. Access Flexibility - The simple X-Y pixel addressing method used in CMOS sensors allows choosing "areas of interest" (windowing). This flexibility is useful in making trade-offs between resolution and frame rate.

4. Reduced cost

Its very clear that CCDs and CMOS detectors have their advantages and disadvantages in certain areas. The most worrying factor when using CMOS sensor in medical imaging application is its relatively low dynamic range and signal to noise ratio. Nevertheless few quality CMOS sensors have reached close to CCD performance in terms of dynamic range and noise figures. Moreover its quite natural that there are trade-offs when choosing one technology over the other.

3.1.4 CMOS image sensor selection

Image quality was given the biggest priority in making the selection of CMOS image sensor. However, because no image quality standard was found for medical cameras, recommendations made by well-known medical institutions, technical papers and existing medical camera data sheets were consulted in making the final selection.
Among many important image sensor parameters that define quality of image, the most important ones include: number of active pixels, dynamic range and colour fidelity.

Number of active pixels

Minimum resolution (number of active pixels) required varies with type of medical application in consideration. For instance American Academy of Dermatology association recommends a digital camera that has a minimum resolution of 1mega pixel for store and forward telemedicine [3]. While a minimum of 0.25mega pixels is recommended for accident and emergency telemedicine [11]. As the goal of this thesis was to design a general purpose medical camera, image sensors which support different capture image resolutions up to SXGA (1280 by 1024) were considered.

Dynamic range

Dynamic range of consumer grade CCD is about 66dB while consumer grade CMOS have 54dB. No specification for dynamic range (Signal to noise ratio) was found for medical application, however sensors with dynamic range of above 60dB are mostly recommended for such applications in manufacturers' data sheets [12, 13].

Table 3.1 below compares specification of seven SXGA (1280 $\times$1024) CMOS sensors from different manufacturers. Data sheet evaluation and comparison was difficult due to missing or omitted specification values, confusing terminologies, and difference in unit of measurement used by manufacturer. In the selection process good dynamic range and SNR were given top priority. From all the image sensors considered Fill Factory's IBIS5 was found to have exceptionally good dynamic range (64dB). In fact for special mode of operation (multiple slope operation) it can reach up to 80 - 100 dB [12]. But the sensor has relatively high dark current and fixed pattern noise. Next choice was OmniVision sensor with 60dB dynamic range and very low fixed pattern noise.

Fill Factory's IBIS5 was available off the shelf while no retail supplier was found for omniVision's OV9620, hence Fill Factory's IBIS5 was selected.

Table 3.1: Image sensor specification comparison

3.2 FPGA

The factors considered in making FPGA selection include:

- Availability of software package for FPGA design, which incorporates VHDL design entry, compilation and logic synthesis, simulation and timing analysis, and device configuration;
- Enough FPGA resources (such as number of logic blocks and size of embedded memory);
- Market availability and cost;
- Speed grade and power consumption.

Altera and Xilinx are the competitive FPGA vendors. In the design however, due to unavailability of the full software package for Xilinx FPGAs , only Altera FPGAs were considered.

First, the required number of logic elements and other FPGA resources in the design were estimated by making a rough design. The preliminary design estimated that an FPGA of at least 5000 Logic elements, 80Kbits embedded memory (minimum) and at least one on-chip PLL were necessary. Implementing this design in FPGAs with no embedded memory would not be efficient because the 80Kbits of memory consume an extremely large number of logic cells. Therefore only Altera FPGAs with on-chip embedded memory were considered.

Among Altera FPGAs, Stratix and Cyclone FPGA devices have on-chip embedded memory. Even though using Stratix devices gives the advantage of having DSP functionality, because of high price and the difficulty in soldering the available package they were not selected.

Cyclone devices offer a targeted set of features optimized for its low-cost architecture. The Cyclone FPGA family consists of four members (shown in Table 3.2), all of which are available in multiple packages for a variety of system and price requirements.

Table 3.2: Cyclone FPGA resource comparison

Of the devices listed in table 3.2 EP1C6 marginally satisfies design resource requirement. Although EP1C12 was the next alternative, which gives more room for later design updates, EP1C6 was selected to keep the price down and get smaller package size FPGAs (as small as 100-pin TQFP).

FPGAs are manufactured for certain target speed grade. The speed grade of an FPGA roughly specifies the propagation delay through FPGA in nano-seconds. For example -3 speed grade implies 3 ns of delay through a level of logic. This specification dictates the maximum FPGA system frequency you can use and it is crucial for good timing performance.

As it will be pointed out in later chapters, the timing performance of speed grade -7 cyclone FPGA was satisfactory and hence a Cyclone EP1C6, speed grade 7 and 144-pin TQFP package was selected.

Cyclone FPGAs use SRAM cells to store configuration data. Since SRAM memory is volatile, configuration data must be downloaded to Cyclone FPGAs each time the device powers up. There are several configuration options including: Joint Test Action Group (JTAG), Passive serial (PS) and Active Serial (AS) configuration schemes. This new scheme is used with the new, low cost serial configuration devices.

In this design a serial configuration device (EPCS1) was selected mainly because of its small form factor and advanced configuration features. Complete data sheet is included in CD appendix (F).

3.3 USB 2.0 Device Controller

The considerations in the selection of the USB 2.0 device controller include enough functionality for the required design task, market availability, cost, and development time required. The length of development time depends on availability of well documented development tools, sample code and easiness of the instruction set or language of the compiler. Although there are many USB peripheral device controllers available in the market, only a few have full support for USB 2.0. In this design the Cypress EZUSB FX2 (CY7C68013), Philips ISP1581, and NetChip NET2280 chips were considered. From these the Cypress EZ USB FX2 controller was chosen. Some of the reasons include:

- Availability of an on-chip 8051 microprocessor, of which the author has some previous experience.
- Availability of complete software solution, library functions, code samples and a hardware development kit.
- Superbly organized detailed documentation, and full web support.

Throughout this thesis the Cypress CY7C68013 will be referred to simply as an FX2 chip.

Chapter 4

Hardware Design Details

4.1 Schematic and PCB Design Layout

Schematic and PCB layout was prepared using Protel Design Explorer. The complete schematic diagram is available in appendix B.

As recommended by the Cypress EZUSB-FX2 data sheet, a 4-layer PCB board was designed. Figure 4.1 shows a photo of the camera board.

Figure 4.1: Photo of the camera board

The PCB board design consists of two boards (front end and main board) connected at a right angle. The image sensor with its associated circuitry is located at the front. A mechanical lens mounting was designed to hold an ordinary 45mm focal length camera lens at proper distance from the image sensor surface.

The FPGA, FX2 chip, power supply circuitry and other additional circuit components are placed on the main board. As a prototype design, the board has several header connectors which were mainly used as test-points for the logic analyzer. These extra pin connections can be removed in the final design.

The camera gets its power supply via the PC USB port, which can supply up to 500mA current at 5V. The board design also provides a dc power jack that may be used for supplying extra power (for example, power for camera lighting). To

conserve power, a switching type 1.5V and 3.3V voltage regulators were used for all digital power in the board. However to improve noise immunity, the analog parts in the image sensor and FX2 chip were supplied by separate 3.3V linear voltage regulators.

4.2 VHDL Design

4.2.1 HDL design approach and choices

HDL programming language choice

There are several high-level hardware design language options for creating an FPGA design. The most common ones are: VHDL (Very high speed integrated circuit Hardware Design Language), Verilog and AHDL (Altera Hardware Design Language).

VHDL is a standard HDL developed by IEEE (Institute of Electrical and Electronics Engineers). The language has two revisions: VHDL'87 (IEEE 1076-87) and an updated version VHDL'93 (VHDL IEEE 1076-93). Because of its more open standard, VHDL is quickly becoming an industry standard for high-level hardware design.

The Altera provided AHDL is limited to Altera devices and is supported only in development tools supplied by Altera. However using AHDL in Altera devices has the advantage of more efficient FPGA resource usage.

Like VHDL, Verilog is a well developed HDL and is supported by many software tools. Nevertheless, in this design, VHDL was selected mainly because of the author's previous experience of it.

VHDL development tools

Altera Quartus II software was used to create the hardware design in VHDL. It provides a complete design environment including HDL and schematic design entry, compilation and logic synthesis, full simulation and advanced timing analysis, and device configuration. Besides Quartus II software provides several third-party synthesis and simulation tools.

Design methodology

In this design a bottom to top hierarchical design approach was followed. First the design was broken down to components and subcomponents. Each component and sub-component was separately built and tested. Finally, all the components were assembled together in the top level design.

The component-by-component design approach gives the benefit of improved code readability, which is crucial to quickly locate VHDL code bugs. Moreover the modular design approach provides easy future upgrading and incorporation of more image processing blocks.

4.2.2 VHDL top level design

The top level VHDL design primarily instantiates seven design components and uses "port map" to connect them.

The building blocks of the VHDL design are:

1. Windowing
2. Bilinear Interpolation
3. Median Filter
4. Colour correction
5. USB Device Interface
6. Timing and Control Logic
7. Clock generation

The timing and control logic component has a generic mapping that provides easy parameter modification. Block diagram representation of the top level design is shown in figure 4.2.

The FPGA is the main camera hardware controller. It handles all image processing tasks and data transfer from the image sensor to the USB device controller. It interfaces to the external devices (Image sensor and USB controller device)

via a set of I/O ports. A table that shows all I/O pin assignments is given in appendix F (CD). Some ports were added purely for testing purposes and can be removed in the final design.

4.2.3 Windowing component

Normally many image processing algorithms use a square sized pixel window (3x3, 5x5, 9x9 e.t.c) as their input. However for efficient use of FPGA resources, a 5x3 window was used. More discussion on the reason for using 5x3 window is given later in this section.

The window component basically creates a window that in effect moves over the raw video frame generated by the image sensor. As it is illustrated in figure 4.3, the component needs to store at least four lines to generate a valid 5x3 pixel window.

Figure 4.3: (a) 5x3 window generation and (b) Bilinear interpolated full-colour image.

Figure 4.4 shows typical hardware implementation of a 5x3 window in a block diagram. The input data is simply moved through the FIFOs. At every FIFO I/O point, the video data is tapped and is clocked in to D type Flip-Flop. Each of the window pixel values are then immediately available at the output of each flip-flop.

Figure 4.4: Block diagram showing typical hardware implementation of 5x3 windowing component

Minimum FIFO depth is determined by the maximum horizontal resolution of the image sensor, which is 1280 in IBIS5 image sensor. Hence, in this design the FIFO was designed to have 1280 depth and 10bit width.

Instances of the required FIFOs were created using Quartus II MegaWizard plugin manager. During compilation, Quartus II automatically implements these

FIFO in memory blocks of 4Kbit size (M4K) of the cyclone FPGA. Even though EP1C6 has enough memory space for implementing these FIFOs, the limitations in using M4K memory blocks may lead to inefficient memory usage.

The first limitation is the memory partition problem, which limits the implementation of several smaller FIFOs in a single M4K. In other words if a memory as small as 1bit is implemented in M4K memory, the whole M4K block will be lost and the remaining memory is not available for other memory implementation purposes. The second limitation has to do with configuring FIFO depth and width. As specified in the Cyclone FPGA data sheet (given in appendix F (CD)), a memory block can only be configured as 4096×1, 2048×2, 1024×4, 512×8 (or 512×9 bits), 256×16 (or 256×18 bits), and 128×32 (or 128×36 bits).

To illustrate an inefficient memory usage, consider implementation of a 1280×10 FIFO. The default Quartus II implementation would make use of five M4K blocks (configured as 2048bit×2 FIFOs) all operating in parallel (sharing the same clock, read request and write request signals). A 5x3 window generation requires four of these FIFOs which means all of EP1C6's 20 M4K blocks would be exhausted just for the Windowing component (which is a very inefficient implementation).

To tackle this problem, a separate component named FIFO_GLUE was designed. Full operation of FIFO_GLUE is presented below.

FIFO GLUE

It is basically used to cascade two synchronous FIFOs to form a logically deeper FIFO. Figure 4.5 shows schematic diagram of the implementation of FIFO_GLUE. The incoming data is continuously written to FIFO-1. When FIFO-1's almost full flag (programmable) is asserted, a read request (RdRq1) is activated. Then after one clock cycle, a write request (WrRq2) of FIFO-2 is activated. Thus data in FIFO-1 gets transferred to the FIFO-2 before it starts to spill over.

On the other hand the reading operation first checks if FIFO-2 has data to be read. If so, the read request (RdRq2) is activated. Otherwise data is read from FIFO-1 by asserting the read request (RdRq1). Complete VHDL code is available in appendix F (CD).

Windowing component top-level architecture

The top-level VHDL code basically instantiates a FIFO_GLUE, M4K FIFOs configured as 1024 x 4, and 256 x 16. Figure 4.6 shows block diagram representation of top level design. All signal buses are labelled and one can trace data flow paths in the design to confirm expected data flow. Complete VHDL code is given in appendix F (CD).

The Empty and Full flags (EF, FF) of all FIFOs are "ORed" together (not shown in figure) to generate composite flags. This ensures that all M4K FIFOS are empty before the composite EF is asserted, and that all M4K FIFOs are full before the composite FF is asserted.

Figure 4.6: Windowing Component top level design

Now having implementation of the window component in mind, the reason for using a 5x3 windows instead of 3x3 can be discussed. As it was discussed in section 2.2, both bilinear interpolation and median filter algorithms require a 3x3 window as their input. The basic way of implementing the algorithms would be to treat the algorithms as two separate processes, each having its own 3x3 window. However, the M4K memory implementation of the FIFOs (for all possible combination of M4K configuration) resulted in inefficient memory usage. To efficiently use memory resources, a 3x5 window was used. As shown in figure 4.3, first bilinear interpolation is performed on the three windows labelled as w1, w2 and w3. The bilinear interpolation results from the three windows were arranged in a 3x3 window, from which median filtering can be done on the fly without a second lengthy windowing operation. The implementation in M4K (figure 4.6) was efficient in terms of memory usage, however due to instantiation of three bilinear interpolation components the design required a bit more logic elements.

4.2.4 Bilinear interpolation

Referring back to figure 4.3, the pixels in a 5x3 window were grouped into three 3x3 windows: W1, W2, and W3. As shown in (b) part of the figure, a bilinear

Table 4.1: Truth table 5 to 3 multiplexer

interpolation performed on the three windows results in a 3x3 full-colour RGB window. This 3x3 window was readily used as an input to median filter.

Performing bilinear interpolation is not a straight forward task. It requires tracking the colours of the pixel surrounding the central pixel. Consider figure 4.3. At blue and red centres, the four neighbouring green sub-samples will be averaged to calculate the missed green value. Similarly at green centres the neighbouring two red and blue sub-samples are averaged to evaluate the red and blue values respectively.

To keep track of the centre pixel's colour, a two bit signal (RC_ODD[1..0]) was defined. The MSB represent oddness of the row number whereas the LSB stands for the oddness of column where the centre pixel lies. For example a pixel that lies on an even row and an odd column will have RC_ODD value of "01".

The block diagram in figure 4.7 shows VHDL implementation of the algorithm. The four arithmetic blocks calculate the average of the neighbouring pixels. Neighbouring pixels around the centre pixel can be located in four different positions (shown in figure 4.7 just below each block). The arithmetic block calculates the average without having to know the colour of the centre pixel. Finally the 5 to 3 multiplexer switches the appropriate average signals (A_1, A_2,A_3 and A_4) to the Red, Green and Blue output signals based on the truth table shown in table 4.1.

The top level bilinear interpolation component basically uses the port map to connect three instances of bilinear interpolation. These three instance perform bilinear interpolation on W1, W2, W3.

4.2.5 Median filter

The median filter basically sorts nine pixel values of a 3x3 window and outputs the middle (fifth) value. After quick look of possible sorting algorithms the algorithm shown in figure 4.8 was selected. The hexagonal blocks represent a simple if-else

statement that assigns the smaller value to the signal output on the left and the bigger value to the right output signal.

D type flip-flops were used to pipeline single data values. The top-level design instantiates three median components. Each component evaluates median value for a one colour window (red, green, blue). The result will be a median-filtered RGB24 video.

The main advantages of this algorithm are the smaller latency (10 clock cycles) that is required to finish the evaluation process and its relatively simple hardware implementation.

Figure 4.8: Median component block diagram

4.2.6 Colour processing

Following the method discussed in section 2.2.3, a 3x3 colour correction matrix was generated using Matlab. A complete Matlab program for generating the matrix is available in appendix F (CD). However due to the limited resources in the FPGA, a full 3x3 colour correction implementation was not possible.

In this design the video output was colour corrected by multiplying the red, green and blue colour element by a factor that ranges between 0 and 2. The VHDL implementation first multiplies the red, green, and blue by byte sized (0 - 255) register values in R_reg, G_reg, and B_reg respectively. Next the multiplied output is divided by 128 by simple shift right bit operation. VHDL implementation is given in Appendix F.

4.2.7 USB device Interface

The USB device interface top-level design instantiates two components: Data flow adapter and Video data buffer. Figure 4.10 shows the USB device interface block diagram representation

Data flow adapter

The data flow adapter was designed to solve data interface width mismatch between the processed 24bit RGB output and the FX2's 16 bit GPIF interface. It receives 24 bit video streaming at 20MHz and outputs 16 bit data at 30MHz. Since both input data rate (24bit x 20MHz) and output data rate (16bit x 30MHz) are equal, no data is lost in the process.

In figure 4.11, a timing diagram is used to describe the operation of the data flow adapter. After assertion of the data valid signal, a 24 bit RGB video is available at every rising edge of the SYS_CLK running at 20 MHz clock. The RC_ODD(0), a signal shows if the current RGB data is from an odd column or even column. Logic low represents for even column and logic high for an odd column. A copy of RGB1 is stored to the RGB2 signal at rising edge of CR_ODD(0) signal (every second rising edge of SYS_CLK). The 16 bit wide signals (BG, RB, GR) couples the newly arrived 24 bit RGB1 data with the previously stored RGB2 in pairs of two (Blue-Green, Red-Blue, Green-Red).

After assertion of DATA VALID2 signal (DATA VALID1 pipelined for two clock cycles) , the paired signals are multiplexed into a 16bit wide data. Multiplexing happens at a rising edge of IF_CLK (30MHz).

Figure 4.11: Timing Diagram of Data Flow Adapter

The Data flow adapter was implemented in VHDL. Complete VHDL code is available in appendix F.

Video data buffer

Basically the Video data buffer component is a FIFO that temporarily stores a single line of a video frame. It also smoothes out the bursty video input and makes it ready for isochronous transmission to the PC, via FX2 chip. From FX2 point of view the rest of the hardware (FPGA and Image senor) is a simple FIFO that will always have valid video data. Section 4.3 on firmware design discusses all the interaction between the FPGA and FX2 chips.

Like the Windowing component, the Video data buffer is implemented as a chain of M4K FIFOs (figure 4.10). All of the embedded memory left over from the windowing component was used to implement a 1792 deep buffer that has 18bits wide data (16 bits of video data and 2bits of video frame synchronization).

Two conditions were carefully observed to obtain a smooth video data transfer using the buffer:

1. The buffer must not get empty. Otherwise the USB isochronous transfer to the PC will hang.

2. The buffer must not overflow. If this happens part of the video frame bytes will be lost and hence the number of bytes in a video frame will be less than expected, which eventually will be rejected by the device driver as bad video frame.

In order to meet those conditions two strategies were planned. The first was to use comparable read and write rates while the second was to synchronize read and write operations for simultaneous buffer read and write.

There is no practical way to control the read rate as it is mainly determined by the Maximum packet size (in bytes) and packets per micro-frame (125μs) set by the PC to meet the frame rate and resolution requested by the user. Hence the focus was turned to controlling the write data rate.

The rate at which the buffer fills is mainly dependent on the rate at which the raw video data is output by the image sensor. Unfortunately the IBIS5 image sensor does not support pixel by pixel reading. The whole line must be read at a rate determined by the image sensor's clock frequency. A possible way around

the problem is to use controlled clock input to the image sensor. However the dedicated PLL clock output used to generate clock input for the image sensor does not allow any operation on the clock line. Moreover the clock frequency of the PLL cannot be below 20MHz. Controlling the clock using external logic would introduce undesirable noise to the system, especially to the image sensor that requires noise free clock input.

The second strategy that writes simultaneously while the PC reads was effective and had relatively less practical implementation problems.

The implementation code uses a firmware generated USB_SOF signal and FIFO LEVEL to determine the time when the image sensor should start reading a new video line. When the USB_SOF pulse arrives an if-else code checks if the FIFO has enough data bytes for the next data packet transfer. If so it will just skip reading a new line from the image sensor. However if the byte number is less, it will allow reading of a new line. The code that makes this decision is in the Timing and Control logic component that controls the image sensor interface. The component is discussed in the next section.

4.2.8 Timing and control logic

All the components discussed so far operate almost passively. Operations that require intelligence are carried out primarily by the timing and control logic component. The logic in this component has a number of responsibilities including:

- Handling of all interface and timing operations to the image sensor.
- Synchronization of video data flow among the modules.
- Receiving new configuration data from the PC (via FX2 chip) and programming the image sensor and FPGA registers accordingly.
- Generation of video frame synchronization bits and error flags in case of hardware or signal error.

The VHDL design is made up of two small processes (Process1_Configuration, and Process2_GlobalCounter) and one large main process (Process3_MainProcess). Each of these processes are discussed in the following sub-sections.

Process1_Configuration

This process is responsible for receiving new configuration data from FX2 and setting a flag to indicate availability of new configuration data to the main process. When the main process finishes executing the new command, it will clear the flag. (refer appendix F (CD) for detailed implementation.)

Configuration data is 16-bit wide. The first 4 MSBs are used in a case statement to uniquely identify the configuration command. A command reference table can be found in appendix C.1.

Process2_GlobalCounter

This process basically is a simple counter that is used to synchronize all operations in the main process. It is implemented as a state machine having three states: counter start, counter stop, and counter reset. The main process uses this counter to exactly determine:

- The number of clock cycles to wait before jumping to next state;
- The pulse width of image sensor input signals;
- Coordinate timing operation and many other operations.

Process3_MainProcess

The VHDL code in the main process is built as a state machine. A simplified flow-chart of complete VHDL code is given in figure 4.12. Each flow-chart box is labelled with a number that indicates the state it belongs to. Below is a brief description of each state:

1. ***Starting_up***: Waits until a start-capture signal is received from the PC, all initialization settings are loaded and are ready to be programmed on the image sensor

2. ***prog_ image_ sensor***: Programs the image sensor with the default settings using parallel programming scheme (IBIS5 data sheet in Appendix F). The same state is called every time a new configuration (eg. changes in resolution, gain, exposure etc.) is detected by *Generate_ Y_ Start* (state that marks start of new video frame). Immediately after image sensor programming is complete, the state machine jumps to either *rolling_ shutter_ setup* or *sync_ shutter_ setup* state depending on the shutter mode selected.

3. ***Rolling_ shutter_ setup***: Sets up image sensor for rolling shutter mode operation

4. ***Synch_ shutter_ setup***: Sets up image sensor for synchronous shutter mode operation

5. ***Generate_ Y_ Start***: marks the start of new video frame capture and it pulses the *Y_ Start* (read new frame) signal. This state is also a strategic state to generate many important signals and to check availability of new configuration data.

6. ***Generate_ Y_ Clock***: This state pulses *Y_ Clock* (read new video line) and carries out a number of decisions based on image sensor and FX2 signal events. It is also a strategic state to take many other decisions.

7. ***Wait_ newline_ newframe***: detects start and end of a line and/or video frame by observing signal events in the output signals of the image sensor. In an event of new line, it jumps to *Generate_ Y_ clock* state. After a few clock cycles the raw video data is written to the FIFOs in the windowing component.

In all of the above operations in the process, the *Global_ Counter* is employed to determine the exact number of clock cycles to wait. In addition it keeps the sequence of operations in a queue.

4.2.9 VHDL design simulation and hardware test

Prior to performing the hardware test, full design verification was performed using AlteraModelSim simulation software. AlteraModelSim proved to be invaluable to quickly and easily find VHDL design bugs.

To be able to verify expected VHDL video processing operation, a test bench code was written based on a technique used in a complete thesis work on FPGA image processing implementation [19]. The test bench code uses a Matlab generated raw RGB image (in binary fromat) as an input and outputs the simulated output image in binary (.bin) format. The conversion of image to and from binary format was performed using Matlab. The test bench and Matlab script files are given in Appendix F (CD).

Figure 4.12: A simplified flow chart of VHDL main process

4.3 Firmware Design

The FPGA does all video processing operations and stores the processed video data in a 16bit wide video buffer. Once the video is ready to be transferred to the PC, the FX2 chip handles all video data transfer following instructions issued to it by the PC.
In this section a firmware code (written in an assembly and C language) that handles the PC instructions is discussed. Prior to explaining the firmware code, first the hardware and software development kits used in this design are discussed briefly.

4.3.1 Hardware and software development tools

Cypress Semiconductor provides a development kit for its FX2 chip, which reduces the development time and learning curve for designing USB device controller firmware. The majority of code development and the testing was done using Cypress FX2 development board (CY3681). The software tools and sample codes contained in the development kit were helpful to quick-start with USB firmware design.

Cypress also provides a library "EZUSB.lib", which greatly simplifies the design problem by providing FX2 register definitions, constants, bit masks, macros, data types, and library function.

An evaluation version of Keil u-Vision 8051 tools was used as firmware development environment. This limited version cannot compile and/or link code size exceeding 4KB. Therefore in this design less important functionalities (such as support for USB 1.x) were left out to keep the code size in the limit of 4K. Moreover instruction codes that end up in a fairly large code size (such as multiplication and division) were replaced by pre-calculated constant values stored in a lookup table.

4.3.2 Firmware design details

The firmware code contains four main source files:

1. The frameworks file (fw.c)

2. Camera descriptor (dscr.a51)

3. Camera main code (MyCamera.c)

4. General Programmable Interface code (GPIF.c)

Although the full source code of each of the above firmware files is given in appendix F, the following sub-sections give a brief description of each of these codes and a few important code implementations.

Frame works

The entire code of the Frame works code is provided by Cypress semiconductor. It handles the basic functionality required by a USB compliant device which include all PC requests, power management, enumeration and re-enumeration.

Detailed code description can be found in an application note included in the FX2 development kit.

USB descriptors

Device descriptor, a formatted data structure which contains necessary camera device information required by the PC, was written in an assembly code. A USB video class standard has recently been defined by USB Implementers Forum (USB-IF), a non-profit corporation founded by the group of companies that developed the Universal Serial Bus specification. The standard gives a detailed description of a standard camera descriptor [13].
Even though a well commentated complete assembly code (dscr.a51) of the descriptor is given in appendix F, some descriptors are worth mentioning and are discussed below.

In the device descriptor section a random vendor ID and product ID were selected, however devices for commercial purposes are required to get a unique vendor ID from USB-IF. These vendor and product IDs must be the same as the ones used

in the device INF file. The operating system will then be able to match a device with its driver.

Regarding the interface descriptor, the USB video class standard requires devices to use an Interface Association descriptor (IAD) defined in IAD Engineering Change Notice (ECN) [13]. The interface descriptor uses two separate descriptors:

- A VideoControl (VC) interface descriptor that fully characterizes all video functions supported by the camera and
- A VideoStreaming (VS) Interface descriptor that contains information that fully describes the video streaming interface.

In this design, however, a single general interface was used mainly because IAD support in windows is still under development.

The descriptor table defines eight alternate settings. Each alternate setting configures end point 2 as an isochronous endpoint for high speed transfer. Except for the difference in the maximum packet size all alternate settings use the same field values. Section 5.3 on device driver design discusses how the maximum packet sizes were selected.

The final descriptors are series of string descriptors that contain a manufacturer name and product name.

Camera main code

This code primarily responds to a set of interrupt service requests and function calls from frameworks.c. This section gives a brief description of code implementation for initialization request, set interface request, vendor command and SOF interrupt request.

Initialization Request:
This routine handles all initialization tasks including:

- End point initialization: End point 2 was selected for quadruple buffered, 1024 bytes sized, isochronous transfer, while other end points except EP0 were set to invalid.

- GPIF Initializations:
- Hardware Initializations: The FPGA's 1.5V and the image sensor's (3.3V and 4.2V) voltage supplies are turned on by asserting the shutdown input. Following successful power up of the hardware, the whole camera hardware is reset by pulsing a 1ms signal on the system reset I/O line. Finally an input signal in the FPGA, that triggers start of image capture is asserted.

Set Interface request
The invocation of this request can be traced back to a change in frame rate or resolution of the video image by the user. When such changes happen, the device driver stops streaming data on the USB bus and sends a request for new alternate setting interface.

Then the firmware code updates the alternate interface and the data packet size that the FX2 chip will commit in the subsequent transfers. In addition few internal variables and a transaction counter (TC) variable that holds number of bytes to be transferred via the GPIF interface are updated. More discussion on GPIF interface is given in the next section.

Another important variable is the FPGA video buffer level which allows the FPGA to control the buffer from being empty or full. This condition is discussed in section 4.2.7.

Vendor Request
This section of the firmware code handles all Vendor specific commands received from PC as described below:

- Initialization request: Programs the image sensor with default settings;
- Set gain: Adjusts the gain of analog amplifier in the image sensor;
- Set offset: Adjusts the offset of the image sensor's analog amplifier. This adjustment is equivalent to adjusting brightness;
- Set exposure(Integration time): Sets the exposure time of the image sensor (related to contrast adjustment) ;
- Set resolution: Sets the resolution;

- Colour correction: Adjusts the Red, Green and Blue content in a video image;
- Zoom: Turns the image sensor to operate in sub-sampled mode (described in the image data sheet);
- Gamma: Turns on/off the on-chip ADC gamma correction.
- Median Filter: Turns on/off the median filter implemented in VHDL.

Each of the above commands along with the command value are transferred to the FPGA by triggering a single GPIF write transaction. (discussed in the section 4.3.3)

Start of Frame (Sof) interrupt

This SOF interrupt sub-routine, asserted every $125\mu Sec.$, is strategically selected to service all isochronous transfer requests. Figure 4.13 shows flow chart of the implementation code. Immediately after assertion of a SOF interrupt, a signal (USBSOF) is pulsed. It is used in video buffer level control (as discussed in sec 4.2.7). The firmware then checks whether EP2 is not full and FPGA video buffer has data ready for transfer. If the condition is satisfied, it goes on to loading synchronization bytes into the first byte space in EP2 buffer. The firmware decides the value of synchronization byte (table 5.4) based on 2bit external input signals from FPGA(refer section4.2.7 for more detail). After synchronization bytes are loaded, transfer of video data follows. The first data packet always contains the synchronization bytes followed by video data, while the second and third data packets (in case double/triple transfer per $\mu Frame$) contain only video data. As soon as the GPIF data transfer is finished (TC = 0), the received data is committed to USB SIE for transfer to the PC.

This section has presented the flow diagram description of the main camera code. The detail code can be found in appendix F.

Figure 4.13: Flow chart representation of ISR_SOF()

4.3.3 General Programmable Interface (GPIF)

The FX2 chip provides two alternative ways for external data interfacing. The first option uses a slave FIFO interface where external glue-logic is used to control data transfer to and from FX2. The second way uses GPIF interface in which FX2 takes control of data transfer through internally coded state-machine. Detailed information on external interface can be found in the FX2 technical reference manual F.

In this design a GPIF interface was selected primarily to avoid implementing external control logic and to exploit its seamless and efficient interface. Details of GPIF operation can be found in the FX2 technical reference manual F.

To simplify the task of creating GPIF interface code, Cypress Semiconductor provides a graphical user interface design tool (Cypress GPIF Designer). Without this tool one would have to do tedious bit by bit coding of waveform descriptor bytes and registers. More information on GPIF designer can be found in the online documentation manual.

The tool is shown in figure 4.14. The first tab page shows logical interface between Cyclone FPGA and FX2, whereas the four tabs following the block diagram page present the graphical editor for defining four distinct waveforms. These waveforms are a 7-state (S0 - S7) state machine which can be configured for a specific data transaction.
In the block diagram page, interface signal names for the CTL and RDY pins of FX2 chip were assigned, interface clock frequency was set to 30MHz and 16bit (2bytes) data transfer width was selected. The description of each signal is given below.

- PLHDR(1..0) are Video frame synchronization bits;
- EOF is a signal that marks end of video frame Tx_Empty and Tx_Full: signals represent the status of FPGA video buffer status;
- TCXpire is a logical signal and it is internally generated in FX2. It is used to flag the expiration of the transaction counter and thus the waveform should stop throttling data to FX2.

- RxWrRq and TxRdRq signals are read and a write request signals respectively
- USBSOF: is a signal indicating the start of a new USB u-frame.

Figure 4.15 shows three waveform configurations used in the design. Only one of the waveforms can be active at a time and is selected by the main camera code next section.

The first waveform (SngWr shown in Fig 4.15 (a) is a single-write waveform. When this waveform is launched, data is written to the FPGA in State number 0 (s0) by asserting RxWrRq and placing data on the bus (Activate data) for one IFCLK cycle. S1 is a decision-point state that enforces an unconditional branch to the IDLE state, which terminates the waveform. Every time a single write waveform is launched, the GPIF engine will cycle through S0, S1, and then IDLE (S7).

The other two waveforms shown in figure 4.15 b) and c) are both FIFO read waveforms. They were split into two waveforms because the design didn't fit into the available seven states.
The second waveform (RDPLHDR) in figure 4.15 (b)) is a waveform configuration for preparing video data on the bus and reserving space for synchronization bytes that will be filled later in the main-firmware. State-0 checks if data is already available on the bus. If no data is loaded, States 1 and 2 read a single word data on the bus by activating Tx_RdRq signal. After one clock period of inactivity (State-3), the state machine enters a special state (FLOW-STATE) highlighted in yellow colour. The special feature of this state is that it provides a way to efficiently throttle data on and off the bus depending on the status of input signals (RDYn). In addition this state continuously loops back until a decision-point state condition is satisfied. This occurs when a firmware loaded GPIF transaction counter (GPIF-TC) expires (TCXpire = 1) . As shown in the flow state condition box (highlighted in yellow), during the FLOW-STATE no GPIF CTL output is activated. The state simply is used to reserve a fixed number of byte spaces (6-bytes in this design) that will later be replaced by proper synchronization bytes in the main camera code.

The third and last waveform (RDVIDEO) shown in figure 4.15 (b) reads the actual video data from the FPGA buffer into End point 2. After one clock cycle period of inactivity (State-0) the state enters to a FLOW-STATE (State-1). The FLOW-STATE condition is shown in the yellow box. Data is read into EP2 only when the end point is not full and the FPGA video buffer is not empty. As indicated in the figure the decision point forces the waveform to exit if the loaded transaction counter expires or the current data is the last data in a video frame (EOF = 1). When TC expires the waveform jumps to S2. Before jumping to Idle state State2 picks the data that is floating on the bus.

Figure 4.2: Block diagram representation of VHDL top level design

Figure 4.5: FIFO_GLUE

Figure 4.7: Bilinear Interpolation

Figure 4.9: Colour Correction

Figure 4.10: USB Device interface top level design

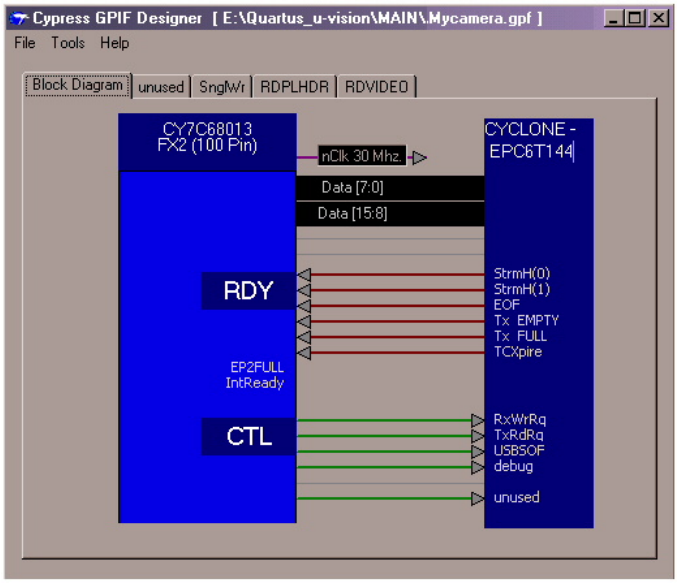

Figure 4.14: Logical interface between Cyclone FPGA and FX2

Figure 4.15: GPIF Waveforms

Chapter 5

Software Design

In this chapter main software design issues and software design details are presented. The chapter is organized in five sections.

The first section (section 5.1) briefly discusses operating system choice. Device driver design matters and details of driver design are covered in sections 5.2 and 5.3. Section 5.2 gives the necessary background information of general device driver development and more specifically USB camera driver development while section 5.3 discusses details of driver design, The last two sections present some concepts of application program development for a video camera (section 5.4) followed by a brief discussion of the design details (section 5.5).

5.1 Operating System

Windows operating system (OS) is a dominant platform for multimedia and many other applications that utilize high performance graphics. However, stability and security concerns with Windows OS tend to prevent its use in applications that require high security and reliability. Other popular operating systems such as Linux offer very good security, however, they tend to lag behind in terms of support for new technology.

For this design project, Windows operating system was chosen mainly because of the author's previous experience on windows. In addition, major motivations to using Windows include:

1. Full USB 2.0 support;

2. Availability of well documented software tools, which include sample driver codes for a range of device types;

3. Free Software development kit for writing an application program.

From the family of Windows OS, Windows XP SP1 was selected primarily to take advantage of the latest resources included in it. Therefore in this report, windows OS is used to refer to Windows XP SP1 professional unless it is specifically mentioned.

5.2 Device Driver: Basic Concepts

Microsoft ships standard drivers with its Windows OS. Fortunate peripheral device designers may find a complete driver suitable for their device. Most often devices which fall in a standard device classes, such as Human Interface device (HID), mass storage device and printers can readily use the built-in class drivers. If a ready to use driver is not available, the next step would be to check if the class driver can be modified by writing additional code to meet device specific requirements. If a device does not fall in one of the standard device classes, device vendors are compelled to write their own driver from scratch. This task is very daunting and requires an in-depth knowledge of windows OS structure.

Microsoft Windows XP SP1 does not include a standard driver for a USB video camera. However, in its Driver Development kit (DDK), Microsoft provides driver sample examples for most common devices including USB video camera. In order to understand these samples, a basic understanding of a Windows driver model and driver layers and particularly a USB camera driver stack is required. Therefore this report will present basic Windows driver concepts with particular focus on topics relevant to the design of a USB video camera driver.

The majority of the contents in this section were sourced from Microsoft driver development documentation. In order to keep the chapter volume small, detailed information is left out but a few important figures and relevant information are included in appendix E.

Windows OS specifies a standard driver model, Windows Driver Model (WDM)

for all devices that work under Windows 98 and later OS. The following subsection gives an overview of this model.

5.2.1 Windows Driver Model (WDM): An Overview

WDM is designed to enable source code compatibility of device drivers across windows operating systems families. The prerequisite for WDM compatibility include support for plug and play (PNP), power management, and Windows Management Instrumentation (WMI).

A Plug and Play (PnP) device must have both hardware and software support that enables a computer system to detect and adapt to hardware configuration changes automatically. The PnP device allows a user to add and remove devices without manual configuration, and without computer hardware knowledge. Plug and Play architecture also provides functions such as hardware resource allocation, loading of appropriate drivers and event notification in cases such as device removal.

Power management adds additional capabilities that allow the system to save power by putting devices in a low power state (sleep) mode when they are not needed. Unlike other systems, which implement power management through BIOS or dedicated monitoring hardware, WDM gives the power management control to the operating system from which the users can set their preferences.

Windows Management and Instrumentation (WMI) provides a standard interface between a device driver and a user-mode application. Using these standard interfaces, an instrumented driver can communicate with the rest of the operating system components, permit configuration of its device, and supply event notifications while it is operational.

Windows OS interacts with a WDM device driver via its discrete modular components, which include the Input-Output (I/O) Manager, Plug and Play Manager, Power Manager, Configuration Manager and Memory Manager. The I/O manager is responsible for handling all I/O operations in a computer system including data flow in peripheral devices. All communication requests between device drivers and the I/O manager are sent via I/O request packets (IRPs). A flow diagram that shows a typical OS and device driver interaction is given in appendix E.1.

In the next section the discussion focus is turned to USB video Camera driver development.

5.2.2 USB camera driver stack

Like all WDM drivers, a WDM USB Video Camera driver follows layered driver architecture.

Figure 5.1: Typical USB camera driver stack

Generally WDM drivers can be grouped into three types: bus drivers, function drivers and filter drivers. Figure 5.1 shows a typical USB video camera driver stack. A general description of each driver contained in the driver stack is given in the next sub sections.

Hardware bus driver

At the bottom of the figure is the hardware bus driver which are responsible for servicing an I/O bus through which a hardware device is physically connected. Microsoft supplies bus drivers for all the major buses (such as PCI, SCSI) as part of the operating system.

Port driver

Just above the bus driver stack is a port driver which consists of a USB hub driver and USB host controller. The USB host controller is implemented as a class/miniclass driver pair. Window XP includes a port class driver (usbport.sys) and three mini-port drivers: universal host controller interface (usbuhci.sys), open host controller interface (usbohci.sys) and an enhanced host controller interface (usbehci.sys), a miniport driver that supports hi-speed USB devices. A USB device may not require all mini-port drivers for its operation.

Most of newly bought PC mother boards include built-in USB hub and a USB host controller. Drivers for PCI USB host controllers are mostly supplied by the individual manufacturers.

Function driver

A function driver is a driver written specifically for a single device or group of devices sharing similar characteristics (device classes). Microsoft provides standard function drivers including a streaming class driver, which supplies system-required but hardware-independent support for streaming devices (such as audio and video devices).

The stream class driver handles application requests conveyed to it by upper level user-mode applications. Depending on the request received, the stream class driver may either handle the requests independently or pass them to lower drivers. Vendor specific requests are passed to a miniclass driver. The miniclass driver (also called simply minidriver), is typically supplied by hardware vendors. In this design, writing a USB camera driver task mainly involves writing code for the minidriver. Further discussion on USB camera minidriver design is given in section 5.2.3.

User mode application and DirectShow filters

All the drivers in the driver stack discussed up to this point run in kernel mode. The application and DirectShow interface shown in figure 5.1 run in user mode. The application is normally a GUI program that receives requests from a user and handles the request with the help of DirectShow interface, a Microsoft supplied streaming media architecture for multimedia applications. It provides easy access to the underlying kernel mode drivers.

A WDM designed video capture driver can readily integrate with DirectShow and use the enormous functionalities supported in it. Hence, video capture task is greatly simplified. DirectShow uses WDM capture filters that send control messages from DirectShow into the streaming class to do most of video capturing operation. DirectShow KSPROXY filter handles all requests from applications. For requests that require kernel-mode services, DirectShow calls the stream class support routine to carry out the request. More discussion on DirectShow and the application program is given in section 5.4.

5.2.3 USB camera minidriver design

Stream class and minidriver interface

The kernel streaming class driver along with other system software components is responsible for handling all video streaming services. Vendor specific processing requests (such as video format, property sets, compression or decompression, and colour space conversion), however are passed to the minidriver that is implemented as a dynamic-link library (DLL)).

After both drivers are initialized, the stream class driver takes control of the request flow. It passes on the requests to the minidriver by means of stream request block (SRB), which contains a command and data associated with that command. The minidriver responds passively to SRBs it received. The interaction between the stream class driver and minidriver is summarized in a block diagram available in appendix E.2.

Most of the work of writing a USB camera minidriver involves writing SRB function callbacks. The minidriver is required to inform the streaming class driver of the function callbacks that are supported in it.

To further simplify the task of writing a USB video camera minidriver, Microsoft provides a USB camera minidriver library (USBCAMD) that handles the majority of streaming tasks.

USBCAMD

USBCAMD is a USB camera minidriver library implemented as kernel-mode DLL. It is designed for streaming video cameras and cameras with snapshot capability (still frames).

Two versions of USBCAMD are available. The newest version, USBCAMD Version 2.0 (Usbcamd2.sys), supports additional features on still pins, power management (e.g. hibernation) and extended versions of the API's.

USBCAMD significantly simplifies the development of USB bus camera minidrivers by:

1. Taking control of all stream class driver and USB bus driver interfaces for the camera minidriver.

2. Handling all streaming tasks and allowing the camera minidriver to extract video frames from the stream in a convenient way.

3. Handling all the synchronization, start, stop, and error recovery issues associated with maintaining the isochronous stream on the USB bus.

All these provisions allow the minidriver writer to focus on support for camera properties and image processing instead of the details of maintaining an isochronous stream on the USB bus.

On its part the camera minidriver is responsible to inform the USBCAMD whether a received data packet is valid, is part of the current video frame or part of new frame. It also handles device-specific requests (such as brightness, contrast etc).

If the camera uses additional non-isochronous streams (such as bulk/interrupt transfer for capturing still image), the camera minidriver is responsible to handle the interface requests.

Figure 5.2 shows a block diagram representation of a driver architecture employed in the design of a USB camera. Except the for camera minidriver (shaded), all the rest of the driver blocks are provided by Microsoft. The USBCAMD library will control the actual connection to the stream class driver and the USB bus driver, thereby simplifying the code contained in the camera minidriver.

Figure 5.2: USB Camera Video capture

USBCAMD can be used in cameras that employ different configuration for video data transfer, sending video frame synchronization, and still frames transfer.

In this design two configuration types were considered:

1. A single isochronous data stream where synchronization information (start and end of video or still frames) is embedded in the data stream. This configuration multiplexes both video and still frames through the same isochronous pipe. It can also use individual video frames as still frames. However, if the camera is required to have a hardware snapshot button (trigger event from hardware), a separate interrupt pipe is used to control and retrieve still frames.

2. A dual isochronous streaming, where one pipe is used for streaming data and the other for synchronizing information (video or still frames). This configuration also allows multiplexing both video and still frames through the same isochronous pipe or reuse individual video frames as still frames.

USBCAMD operation

When a USB camera is plugged in to a USB port, the PnP detects the camera and calls the DriverEntry of the device minidriver. In its DriverEntry routine, the camera minidriver binds to USBCAMD library by calling USBCAMD_DriverEntry. USBCAMD will then register the minidriver with a stream class driver. From here onwards the stream class driver sends its requests to the camera minidriver via stream request blocks (SRBs) structure. Except for a few SRBs, including SRBs that report video format and SRBs that handle property requests, SRB requests received by camera minidriver are passed on to USBCAMD for processing.

Depending on the SRB requests passed to it, the USBCAMD will start or stop the isochronous stream. While processing SRBs, USBCAMD calls back the camera minidriver for device specific tasks via a set of callback routines that are passed to it in USB_DEVICE_DATA2 structure. The device specific tasks include bandwidth allocation (selection of appropriate alternate setting) and video format selection.

A detailed SRB request handling by camera minidriver and USBCAMD is given in appendix E.3.

Once all initialization and setup SRBs are processed successfully, the isochronous streaming of video data is started. The interaction between USBCAMD and camera minidriver happens in two situations:

The first condition occurs when new data is available to USBCAMD from USB bus driver (USBD). At this time, USBCAMD calls the camera driver's CamProcessUSBPacketEx routine which is processed at low Interrupt request level (IRQL = DISPATCH_LEVEL). In this routine the camera minidriver sets appropriate error flags in the case of error conditions. It must also set a new frame flag if the received packet marks the beginning of a new frame.

The second condition, which is processed at high interrupt request level (IRQL = PASSIVE_LEVEL), happens when a video/still frame transfer is completed. At this point USBCAMD calls the camera driver's CamProcessRawVideoFrameEx routine. This is the chance for the minidriver writer to implement any kind of image processing including, raw video data decoding, colour space conversion, compression/decompression on to the newly received video frame. An error code is returned if there is insufficient frame data or an error occurred while processing the video frame.

Design procedures and software tools employed in the design of a camera driver are discussed in the next section.

5.3 Driver Design Details

As it was mentioned in chapter 4, designing a device driver from scratch is a very daunting task that requires in-depth knowledge of windows OS structure. Luckily Microsoft provides sample codes for most common device driver types including a sample for a USB video capture device. The sample was designed by Microsoft to support two Intel USB digital camera models: YC76 and YC72. It was written to comply with WDM video capture requirements and the USBCAMD specifications. Throughout the rest of this report the sample will be referred to as USBINTEL.

In this design USBINTEL source code was modified to support some specific requirements in the camera design. USBINTEL code is extremely big, basically thousands of lines written in C programming language. Although the code is well commentated, no code documentation is provided. Hence; understanding the basic code framework, data structure and functions and line by line understanding of the source code was essential. The next sub-sections give a brief description of:

1. Code modification
2. INF file and driver installation
3. Driver debugging and testing

	USBINTEL	MyUSBCAM
1	Uses dual isochronous stream for video data stream and synchronization information	Use a single isochronous data stream where synchronization information (start and end of video or still frames) is embedded in the data stream.
2	video image size of: 320x240, 160x120, and 176x144.	Maximum packet size supports video image size of: 320x240, 160x120, 176x144, 640x480, 800x600, 1024x768 and 1280x1024
3	Video image format is in YUV1 format	Video image format used is RGB24
4	USB 1.0 cameras	USB 2.0 camera
5		

Table 5.1: Summary of major driver source code modification

5.3.1 Driver code modification

Although it is difficult to discuss every line of code modified, the major modifications in the design will be discussed. Table 5.1 gives a summary of differences between USBINTEL and the modified driver code. This report will refer to the modified code as MyUSBCAM.

The code modification for the differences given in the table are explained one by one in the following sub-sections.

Support for high resolution and frame rate images.

When the camera user changes the resolution and/or frame rate, the camera minidriver calculates the isochronous channel's maximum packet size that satisfies the requested frame rate and resolution. That means the support for the selected resolution and/or frame rate depends on the available maximum packet sizes in the alternate settings of the USB isochronous transfer. As a result of this, each of the supported video resolutions in MyUSBCAM can only be captured at predetermined frame rates. For example if one wants to have 5, 10, 15, 20 and 25 frame rates for a VGA resolution video, at least five alternate settings (each

with a maximum packet size that satisfies the frame rate) are required. In this design, due to the 4K code limitation in the KIEL firmware development, only seven alternate settings were realized.

The following important conditions were taken into account when making the selection for maximum packet size for the seven alternate settings:

1. The maximum packet sizes selected should as much as possible satisfy the possible resolution/frame rate options.
2. As it was pointed out in the previous chapter, video synchronization bytes are sent at the start of every USB packet. Therefore it becomes necessary that every full video frame be transmitted in non-fragmented packets of data.
3. The frame rates for each resolution should, as much as possible, fall into the common frame rates, such as 10, 20, 25, 30.

After making several trial and error calculations using a Microsoft Excel worksheet, the maximum packet size values that best satisfy the above conditions were selected. Table 5.2 shows the possible frame rates for each of the supported video resolutions.

The example below illustrates maximum packet size calculation for an SVGA (800 x 600) 24bit RGB video at 25frames/sec refresh rate:

$$Video\,Data\,Rate = 640 \times 480 \times 24 \times 25$$

$$Video\,Data\,Rate = 184.32 Mbits/sec = 23.04 MBytes/sec$$

$$Bytes/\mu Frame = 23.04 MB/sec \times 125 \mu Sec. = 2880 Bytes = 3 \times 960 Bytes$$

$$Maximum\,DataPacket = 960 Bytes$$

Table 5.2: Maximum Packet size for every possible resolution and frame rate

Table 5.3: AvgTimePerFrame, MinFrameInterval, MaxFrameInterval calculation results

Once the maximum packet sizes were selected and the possible frame rates for each resolution are determined, the next step was USBINTEL code modification.

The camera minidriver is required to report the video format (width, height, frame rate, colour space etc) it supports to the DirectShow interface in a data structure called KSDATARANGE VIDEO. The modification in this regard was to write additional KSDATARANGE VIDEO structures that indicate support for additional video resolutions and frame rates (given in table 5.1). KSDATARANGE VIDEO contains two structures in it: KSDATARANGE and VIDEO STREAM CONFIG CAPS. In the KSDATARANGE, colour space was specified as RGB24. Whereas in the VIDEO STREAM CONFIG CAPS structure the minimum, average and maximum frame rates were specified. The frame rates were converted into 100ns units of frame interval as given in equation 5.1.

Table 5.3 shows minimum, average and maximum frame intervals for each of the possible frame rates.

$$Frame\,Duration = \frac{10,000,000}{FramePerSecond} \tag{5.1}$$

Using single isochronous data stream

USBCAMD calls the camera driver every time a data packet is available from the USB bus driver (USBD). In its routine the camera minidriver must set a new frame flag if the beginning of a new video frame is detected. It must also set appropriate error flags in case of error conditions.

In this design the minidriver routine was modified to examine the synchronization byte (syncByte) embedded in each packet received. The interpretation of the

Table 5.4: Synchronization byte values and their meaning

synchronization byte is shown in table 5.4.

As discussed in section 4.3 the first byte in each data packet holds the total size of the payload header. This value was assigned to ValidDataOffset variable, which the USBCAMD uses to determine the offset from which the buffer should be copied. This eliminates the extra buffer copy.

At a completion of each video frame, USBCAMD calls another camera driver routine (CamProcessRawVideoFrameEx). This is the chance for the minidriver writer to implement any kind of image processing. However in this design nothing is done except copying the received frame buffer to a new buffer because an attempt to perform certain image processing in the driver created driver malfunctioning.

Finally the copied video frame is directly accessed by DirectShow and is displayed on an application video window.

5.3.2 Installation

When windows detects a new device in its bus, it searches for the proper device driver from its driver database using the device's Vendor ID and Product ID. If no match is found, it prompts the user to provide the device installation package. The installation package should at least include an INF (A device setup information file) and one or more device drivers. Detailed information on an INF file and installation packaging can be found in Windows DDK documentation, however in this section only the steps and tools used are discussed.

The INF file for USBINTEL was used as template and each field in the INF file structure was written with the aid of Windows DDK documentation. To keep the volume of this thesis small and make it readable, all detailed INF code is given in CD appendix F. Due to lack of previous experience and very tight INF file syntax rules, a significant development time was required. However, an INF checking tool (chkinf.exe) supplied in the Windows DDK, was helpful in debugging syntax errors.

5.3.3 Debugging and testing

Kernel-mode driver debugging was done using Microsoft's Windbg tool. Kernel mode driver debugging requires two PCs: a target computer and a host computer. The target computer is the PC which runs the kernel-mode driver. The host computer is the PC which runs the debugger. The debugging tool was extremely useful in finding many bugs. For more information on performing kernel mode driver debugging consult online documentation of Windbg.

After driver development and INF file writing tasks are completed, it is good practice to test them. Microsoft provides Windows Hardware Quality Labs (WHQL) tests via its website to test drivers for compliance with requirements of the Windows Logo Program for hardware.

5.4 DirectShow Interface

Writing an application program for a video device that confirms with WDM requirement, is greatly simplified when using a Microsoft DirectShow interface (a software component in Microsoft DirectX). DirectShow is basically designed for multimedia purposes and it offers efficient multimedia processing, support for variety of video formats, synchronization of audio and video in multimedia devices and support for a variety of video input sources.

DirectShow uses media streaming architecture, which allows the use of high-level APIs, and uses plug-in components called filters. Filters separate the processing of digital media data into discrete steps; each filter represents one (or sometimes more than one) processing step. A Video Mixing and Rendering (VMR) filter, MPEG compression Filter, and file reading filter are some examples. Third parties can create their own filters to perform any type of custom processing.
The majority of the contents in this section were sourced from Microsoft DirectX online documentation.

5.4.1 DirectShow filter and minidriver communication

DirectShow APIs are accessed through Component Object Model (COM) interfaces. The Component Object Model (COM) is an object-oriented programming

model used by numerous applications. Only basic knowledge of working with COM objects is required for application program development, however in-depth knowledge of COM programming is required when writing one's own filter component.

Figure 5.3: DirectShow Interface

DirectShow application is written by creating an instance of a high-level COM object called the filter graph manager. The filter is then used to create configurations of filters (called filter graphs) that work together to perform a desired task on a streaming media (Video capture and display in this design). The filter graph manager and the filters themselves handle all buffer creation, synchronization, and connection details, so the application developer needs only to write code to build and operate the filter graph.

Microsoft provides a GraphEdit tool, which is useful to test and experiment any filter graph before writing an application code.

Figure 5.4 shows USB Video Camera filter graph built using GraphEdit tool.

Summarized steps on how to build filter graph for WDM USB video camera is given below.

1. As soon as the camera is plugged into the USB port, the System Device Enumerator registers the device on a user's system, and puts it under the WDM streaming capture devices category. In a DirectShow filter graph, any WDM capture device appears as the WDM Video Capture filter. The WDM Video Capture filter configures itself based on the characteristics of the driver. It appears under a name provided by the driver.

2. To select a capture device from enumerated devices, System Device Enumerator is used. The System Device Enumerator returns a collection of device monikers, selected by filter category. (A moniker is a COM object that contains information about another object, which enables the application to get information about the object without creating the object itself.

Figure 5.4: Filter graph of USB Video Camera in GraphEdit tool

Later, the application can use the moniker to create the object. For more information about monikers, see the documentation for IMoniker in the Platform SDK) . The selection of a particular capture device from the list returned by the Moniker is done by searching the list for the matching device name (technically, the friendly name of the device), in this case My USB Video Camera.

3. Once the device is selected (pointer to the device is obtained), it is added to filter graph using AddFilter method. As it has been pointed out earlier in DirectShow, devices are added to filter graph as a filter with INPUT and OUTPUT pins.

4. To simplify the task of building capture graphs, DirectShow provides a helper object called the Capture Graph Builder. The Capture Graph Builder exposes the ICaptureGraphBuilder2 interface, which contains methods for building and controlling a capture graph.

5. After adding all the required filters to the filter graph, there are two options to build a complete video capture graph. The first way is to connect the filter pins one by one and another way is to use Intelligent Connect. All connections are "intelligent," and additional filters may be added to the graph as needed.

6. The final step is to render the graph using render method exposed by ICaptureGraphBuilder2.

In addition to playing Windows Driver Model (WDM) video capture devices, DirectShow provides enormous multimedia tools including support for various property sets (such as brightness, contrast, saturation, and so forth), video mixing and a lot more.

5.4.2 Programming language and programming environment

DirectShow is designed primarily for C^{++} development. One can also use C to develop DirectShow application. Application development using Microsoft Visual Basic is also possible; however access to DirectShow API is limited to a small subset. For this design, C^{++} was chosen primarily because of the extensive C^{++} support and documentation provided in the DirectX Software development kit.

Microsoft Visual C^{++} was selected as programming environment because of availability of good documentation and code samples written by Microsoft Visual Studio in the software development kit (SDK).

5.5 GUI Design

As it has been discussed in chapter 4, the application program uses DirectShow interface to communicate with the kernel mode USB camera driver. This section discusses the application program code that was designed to display the video captured and bring complete control of the camera to the user.

The GUI was developed using Microsoft Visual C^{++} 6.0. A dialog based GUI was created using Microsoft Foundation Class (MFC) application wizard. The wizard creates an application shell where the C^{++} code is to be written. The project was set up to include DirectShow link libraries (the stmiids.lib and quartz.lib).

The application window layout is shown in figure 5.5. The following controls are available to the user:

- Start, pause, stop the streaming video;
- Change video frame rate and Resolution;
- Slider controls for adjustment of brightness, contrast, exposure and RGB colour balance;
- Push button controls used for turning on/off Zoom, gamma correction and median filtering.

Figure 5.5: GUI

The C^{++}code consists of two main classes: CMyCameraDlg (dialog class) created by MFC application wizard and a newly created MyCamera class.

The dialog class basically monitors events in the application window and uses methods defined in MyCamera to perform the tasks requested by the user. In the dialog class static controls and member functions were added. Member variables were also declared and initialized. Complete code is included in appendix F.

The MyCamera class handles all DirectShow interface details through its public methods.
Although complete MyCamera class code is given in appendix F, each of the capture tasks listed below are discussed in logical order.

1. Device Enumeration
2. Initialization
3. Setting Capture property
4. Controlling the stream capture

Device Enumeration

As it was discussed in chapter 4, System Device Enumerator was used to detect connection of MyUSBCAM. The camera was uniquely identified by its FriendlyName (a name specified for DisplayName field in the camera's INF file). An arbitrarily name was selected for the device as USB VIDEO CAMERA MX. During enumeration a function (*EnumMyCamera()*) sequentially searches through all connected video capture device's FriedlyNames. If a FriendlyName that matches MyUSBCAM's FriendlyName is found, the function immediately binds the camera device to the filter pointer (m_pMyCam). The filter pointer is used by the rest of the class methods to perform a range of video capture operations.

Initialization

During initialization the GUI basically prepares a filter graph by creating an instance of filter graph manager (m_pGraph). First MyUSBCAM is added to the filter graph by using the filter pointer (m_pMyCam) obtained from EnumMyCamera(). The next step was to add the Video mixing & Rendering filter (VMR) available in DirectShow filters. The VMR filter uses the graphics-processing capabilities of the computer's display card to perform video rendering. It does not use the host processor, because to do so would greatly impact on the frame rate and quality of the video being displayed.

The default mode of VMR uses a windowed mode, in which the renderer creates its own window to display the video. However, in this design, a Windowless mode was used so that the video is displayed on the application window (created by MFC dialog wizard). Besides to keeping the GUI in a single window, windowless mode gives the advantage of using an own window style and title.

After the VMR was initialized for windowless rendering mode it was added to filter graph.

Setting Capture Property

Setting the capture property includes video resolution and frame rate as selected by the user. In order to do this, first a pointer to the filter's capture pin was obtained. This pointer was used to search for the VideoInfoHeader structure, where the frame rate and video resolution values are contained. Next the VideoInfoHeader was updated with new values and finally a DirectShow function (IAMStreamConfig::SetFormat) was used to set these new values.

Setting Video ProcAmpProperty

ProcAmpProperty refers to image adjustment settings such as brightness, contrast, hue, saturation etc. The settings can be done either using camera hardware (if supported) or using the PC's graphics adapter.
As it was discussed in section 5.3 (driver design), the camera driver exposes an IAMVideoProcAmp interface which enables an application to adjust the qualities of an incoming video signal, such as brightness, contrast, hue, saturation, gamma, and sharpness.

These adjustment settings can also be performed using VMR9. The VMR performs these settings using the graphics adapter. However in this design only camera hardware adjustments were implemented.

The implementation of these settings is done indirectly by changing the image sensor's amplifier settings and values of a few registers in the FPGA.

The Video camera device filter exposes the IAMVideoProcAmp interface which enables an application to adjust video signal qualities, such as brightness, contrast, hue, saturation, gamma, and sharpness.
Depending on the slider which initiated the function call, new VideoProcAmp values were set using methods exposed by the IAMVideoProcAmp interface.

Controlling capture process (Start, Pause and Stop)

These functions control the streaming median via the DirectShow media control. A pointer to this control (m_pMC) was obtained by queryinterface method (exposed by capture graph pointer).

These functions start, pause or stop video capturing based on user commands issued from the GUI.
In the design DirectShow Event Handling was not implemented. For complete application program code refer appendix F.

Chapter 6

Results and Discussion

This chapter covers results of the whole design process. It starts with hardware results including power consumption, FPGA resource usage, and logic analyzer plots of important interface signals. Next software (driver and application program) performance results are presented. Finally video and still image results are presented. Before concluding the chapter, a few unresolved problems are discussed.

6.1 Hardware

6.1.1 Power consumption

Table 6.1 shows power consumption of each of the main IC chips and the total power drawn from a USB port.

Table 6.1: Camera power consumption

Even though the steady state current consumption is 260mA (far below the 500mA maximum current a USB port can deliver), at start up the camera draws about 480mA current due to the following reasons:

- High power-up current of the Cyclone FPGA (500mA @ 1.5V);

- High current consumption of image sensor at start-up which lasts until system clock is applied.

6.1.2 FPGA resource usage

Table 6.2 gives a summary of total FPGA resource usage by each design component. Almost all the FPGA resources were exhausted (87% logic cells and 90% of memory bits). Although the statistics for memory usage is 90%, practically all M4k memory blocks are 100% used. Hence there is no room for extra memory implementation. The bilinear interpolation, colour correction and Timing/Logic control components used low percentage of the total logic cells and did not use memory resource at all.

The total I/O pin usage was 75% (74 out of the available 98 I/O pins).

Table 6.2: FPGA Resource Usage

Although Cyclone FPGA is generally an excellent target device for the project, the FPGA used in the design (EP1C6) was small and fitting the design required a number of design iterations.

6.2 Timing Results

The FPGA timing performance is summarized in table 6.3. The FPGA speed grade (7) was fast enough to satisfy all timing requirements in the design which was running at clock frequency of 20MHz (50ns period) and 30MHz (33.33ns period). The timing results shown in table 6.3 were quite good to ensure proper timing operation.

Table 6.3: FPGA timing performance

The next subsections present and discuss logic analyzer plots of important interface signals taken while the camera was capturing CIF video at 20fps.

6.2.1 Image sensor - FPGA main interface signals

Figure 6.1 shows logic analyzer snap-shots of main interface signals. PXL_OUT is the image sensor's analog video output while Y_START and Y_CLOCK are FPGA output signals which command the image sensor to start reading a new video frame and a new line respectively. The PXL_VLD and LAST_LINE signals are the image sensor's output signals which indicate the start of valid video on PXL_OUT and the end of a video frame respectively.

Figure 6.1: Main interface signals in-between image sensor and the FPGA

All of the signals were free of glitches, which otherwise would make the image sensor to malfunction. The results were checked against timing specification of the image sensor input signals. It was verified that enough setup time (>10ns) and hold time (>15ns) is available for each of these signals.

To have a closer look at the image sensor analog output, figure 6.1 (b) shows a magnified plot of a small section (marked with dotted line) in figure 6.1 (a).

A further magnified plot in figure 6.1 (c) clearly shows the time at which A/D conversion of analog image sensor takes place. It was made certain that A/D conversion takes place at the peak value of analog signal swing.

6.2.2 FPGA - FX2 interface results

The logic analyzer plots in figure 6.2 shows the interface signals between the FPGA and FX2. The USBSOF signal is pulsed periodically at 125μs interval. At this point the VHDL code makes sure that enough data bytes are available for the next USB transaction. Otherwise it asserts Y_CLOCK signal for a new line read.

The RDRQ signal is asserted after a few clock cycles determined by the execution time elapsed before launching GPIF read. EOF, PLHDR0, PLHDR1 are video

Figure 6.3: Frame rate results

synchronization signals. Depending on the status of these signals, the firmware appends appropriate synchronization byte.

The three figures in part a), b) and c) show single, double and triple packets per u-frame transactions respectively. The timing results shown in figure 6.2 verify expected operation.

Figure 6.2: Main interface signals between the FPGA and FX2.

6.3 Overall Software Performance

This section presents overall software performance including, video frame rate for every resolution, isochronous channel performance and PC resource usage (CPU time and memory). First results for the driver that uses single isochronous channel (explained in section 5.3) is presented and results of the second driver which uses double isochronous channel will follow.

6.3.1 Single isochronous channel

Frame rate

Table 6.3 summarizes frame rate achieved for each resolution. The lightly shaded video frame rates are those which were successfully played. Those with no shading were not implemented because either they are too high or too low. The darker shaded frame rates were not achieved due to the small FPGA video buffer size (discussed in section 4.2.7).

Bad packets

The camera driver analyzes a received packet to decide whether a data packet is good or bad. If the data packet contains a smaller number of bytes than expected or if the payload header indicates an error signal, the video frame is dropped.

As the results in table 6.3 show no frame was dropped during the video capture. Therefore it can be safely concluded that there were no bad packets.

CPU and memory Usage

On a 2.6GHz Intel Pentium machine, the average CPU usage by a USB device driver and application program was only 12%. This CPU percentage usage remains almost the same for the various data transfer rates (1.15MB/sec. to 23.04MB/sec).

6.3.2 Double isochronous channel

The frame rate results of the double isochronous channel driver were the same as the previous results in single isochronous. However the CPU time usage was five times more (about 60 %) compared to the 12% CPU time usage in the single isochronous channel. The main reason is because now the PC is required to service two isochronous channels in a single μ-Frame.

6.4 Video/Still Image Results

This section presents the snap-shot photos of the video image. An MPEG format sample video clip is given in the CD appendix (F). Even though the camera can capture up to 1024×768 resolution at 10fps, due to slow hard disk writing speed only up to a a 640 x 480 (VGA) at 10fps video could be captured. A DirectShow sample (AMCAP) included in Microsoft DirectX SDK was used to capture the video to a file.

Although no digital video quality measurement will be done (as it is out of the scope of this thesis), simple analysis on the results of colour correction will be presented. In addition a subjective comment will be given on a skin photo taken by the camera.

Figure 6.4: ColorChecker results

Table 6.4: Colour comparison

Figure 6.5: Comparison

Colour correction results using ColorChecker

Figure 6.5 shows result of colour correction. A Matlab generated image of the colour chart and a photo taken by Sony camera are also presented for comparison. A simple subjective observation on the colour correction results clearly shows that the colour correction operation has made significant improvement; however there is still a considerable colour deviation from the original ColorChecker colours.

Table 6.4 gives the RGB values for each of the colours in the ColorChecker photos shown in figure 6.5. Figure 6.5 graphically displays comparison of red, green and blue values with the original values in the ColorChecker. The values were first sorted in ascending order based on the RGB values in the ColorChecker.

Real-life photos

The image in figure 6.6, although it does have some colour deviation it shows good contrast and sharpness.

6.5 Unsolved problems

SXGA (1280 x 1024) resolution:

When capturing an SXGA resolution video, the FPGA video buffer (discussed in sec 4.2.7) becomes full and some video data and synchronization bits are lost. Consequently, the video frame synchronization is lost and each frame is rejected by the driver as bad frame.

Figure 6.6: Palm photo

The problem could easily be solved by using a larger video buffer. However, due to shortage of FPGA M4K memory blocks the problem was left unsolved.

Mirrored (laterally inverted) video image:

The orientation of the image captured depends on the lens used, the image sensor's pixel output sequence, and on how DirectShow finally draws the image on the PC screen. The default DirectShow display resulted in a laterally inverted video image. Different solutions to the problem were sought. Optical solution using a lens arrangement was technically difficult and nothing was possible to change the image sensor output sequence. Hence, image flipping was done in DirectShow; however, flipping higher data rate videos (9.22 MB/s and above) was unsuccessful.

Chapter 7

Conclusion and Recommendations

This thesis report has presented a detailed design and practical implementation of a USB 2.0 video camera. The majority of the goals set at the outset of this project have been successfully met; however, future work is required to make the camera available for telemedicine applications. All in all, this thesis can be considered a success. The knowledge and experience gained from this project will certainly be helpful in future camera designs.

Conclusions for hardware design, software design and the video image result are given one by one in the following sub-sections.

Hardware design

The important decision to put all image processing tasks in the FPGA was crucial to the success of this project. However, the limited memory and logic resources in the FPGA (EP16C) brought additional design challenges. Using bigger FPGA with DSP functionality in future work will definitely enable the designer to implement additional image processing operations such as gamma correction, full white balance, sharpness and other image enhancement algorithms.

The VHDL code development, simulation and actual testing for implementation of image processing as well as generating the necessary interface and control signals in the FPGA, demanded a lot of time and hard work. The final result, however, was rewarding.

The Cypress EZUSB-FX2 USB 2.0 device controller was a perfect choice for

the project even though the unavailability of full-version firmware development software hindered the exploitation of the rich functionality in the chip.

Software design

The choice of windows as operating system platform and the subsequent decisions for using a WDM camera driver were crucial for the successful end of this project. The enormous software tools and documentation provided by Microsoft were integral to the success. In addition the full integration of WDM with DirectShow and streaming architecture contributed largely to the very efficient video capture design.

Additional future work in software could include still image acquisition and video image capture to a file. Real-time capture of video images would however demand a fast computer and a fast hard drive.

Video image results

Even though more in-depth knowledge of imaging science is required to evaluate the quality of the video/still image result, generally the video image quality is satisfactory. However due to poor colour reproduction of the image sensor, further work will be required to solve the problem. Even though the 3x3 matrix colour correction has improved the overall colour response of the camera, further investigation is required to obtain good colour reproduction.

For better results in colour reproduction and image sharpness future design may consider using the new Feveon image sensor which is capable of capturing full-colour image. The new FillFactory's IBIS5A image sensor (pin compatible with the older IBIS5) could also be easily tested on the same camera board to check for improvement in colour response and noise.

Bibliography

[1] Constantine-Basil Prouskas , "Telemedicine: Today and Tomorrow." August 2004, http://www.doc.ic.ac.uk/

[2] Malcolm Pradhan, "Important Concepts in Telemedicine." 1994, http://camis.stanford.edu/people/pradhan/articles/telemed.html

[3] American Academy of Dermatology Association, "Position Statement on Telemedicine." October 2004, http://www.aadassociation.org/telemedicine.html

[4] Nandor Balogh, Vytenis Punys, Jurate Puniene, Jonas Punys "Recommendation on the use of lossy compression in clinical environment" May 2003, http/www.samta.offis.de/projekte/samta/samta-d10.pdf

[5] Perdnia, D. A., Gaines, J. A., and Russum, A. C., "Variability in physician assessment of lesions in cutaneous images and its implications for skin screening and computer-assisted diagnosis." 1992. http://www.ncbi.nlm.nih.gov/entrez/

[6] Eastman Kodak Company, "CCD sensors." May 2001, http://www.kodac.com/go/ccd

[7] Albert J.P. Theuwissen, "Solid-State Imaging with Charge-Coupled Devices." 1997, AA Dordrecht Kluwer Academic Publishers, Netherlands.

[8] D. Lotwiller, CCD Vs. CMOS: "CCD Vs CMOS Facts and Fiction" January 2001, http://www.dalsa.com/markets/ccd_vs_cmos.asp

[9] Nicolas Blanc, Zurich, CCD Vs CMOS - has CCD imaging has come to an end, http://www.ifp.uni-stuttgart.de/publications/phowo01/Blanc.pdf

[10] James Janesick, Sanoff Corp., "Dueling Detectors CMOS or CCD ?", February 2002, Spies magazine
http://oemagazine.com/fromTheMagazine/feb02/detectors.html

[11] Jonathan Benger etal, "Image effects on the accuracy of telemedicine"
http://www.catchword.com/rsm/

[12] Fillfactory Image Sensors, IBIS integrating CMOS imagers,
http://www.fillfactory.com/htm/products/htm/ibis.htm

[13] Zoran Micorelectronics Ltd, Product Brief, http://www.zoran.com

[14] USB implementors forum, "Universal Serial Bus Device Class Definition for Video Devices." 2003, http://www.usb.org

[15] Tommy Olsen and Jo Steinar Strand, "An improved image processing chain for mobile terminals." May 2002,
http://siving.hia.no/ikt02/ikt6400/g23/Report.htm

[16] Rajeev Ramanath, Wisley E Synder, Griff L.Bilbro, "Interpolation Methods for the Bayer colour Array." July 2002, Journal of Electronic Imaging

[17] "An entirely new way to capture color" 2004,
http://www.foveon.com/X3_tech.html

[18] Philippe Longère, Xuemei Zhang, Peter B. Delahunt, David H. Brainard, "Perceptual Assessment of Demosaicing Algorithm Performance." January 2002, Proceedings of the IEEE, vol. 90, no. 1, 123

[19] Jan Axelson, "USB Complete 2nd edition." 2001, Madison USA.

[20] Anthony Edward Nelson, "Implementation of Image Processing Algorithms on FPGA Hardware." May 2000
http://www.hcs.ufl.edu/~saleh/rcliterature/
1, 3

Appendix A

Data Sheet Summary

27

A.1 Image Sensor (IBIS5-1300c)

3

3

IBIS5 Pixel Characterstics

23, 25

xi, 23

xi, 23

xi, 24

Electro-Optical Specification

25, 26

IBIS5 spectral response curve

IBIS5 Timing Diagrams

A.2 Cyclone FPGA (EP1C6T-144)

A.2.1 Features

- 5,980 LEs.
- 92,160 RAM bits
- Supports configuration through low-cost serial configuration device
- Support for LVTTL, LVCMOS, SSTL-2, and SSTL-3 I/O standards

- Two PLLs provide clock multiplication and phase shifting
- Eight global clock lines with six clock resources available per logic array block (LAB) row
- 98 I/O pins

Device block diagram

Memory configuration sizes

The memory address depths and output widths can be configured as 4096×1, 2048×2, 1024×4, 512×8 (or 512×9 bits), 256×16 (or 256×18 bits), and 128×32 (or 128×36 bits). The 128 x 32- or 36-bit configuration is not available in the true dual-port mode. Mixed-width configurations are also possible, allowing different read and write widths. Table A.1 summarize the possible M4K RAM block configurations.

Table A.1: Possible M4K RAM block configurations

A.2.2 Altera serial configuration device (EPCS1) features

- 1Mbit flash memory devices that serially configure Cyclone FPGAs
- Four-pin interface
- Non-volatile memory
- 3.3-V operation
- Re-programmable memory with more than 100,000 erase/program cycles
- Programming support with ByteBlaster II download cable
- Software design support with the Altera Quartus II development

A.3 EZUSB-FX2 (CY7C68013-100pin)

A.3.1 Main features

- Single-chip integrated USB 2.0 Transceiver, SIE, and Enhanced 8051 Microprocessor
- Software: 8051 runs from internal RAM, which is downloaded via USB or loaded from EEPROM
- Four programmable BULK/INTERRUPT/ISOCHRONOUS endpoints with buffering options: double, triple and quad
- 8- or 16-bit external data interface
- Master or slave operation: FIFOs can use externally supplied clock or asynchronous strobes
- General Programmable Interface (GPIF) which allows direct connection to most parallel interfaces; 8- and 16-bit
- Integrated, industry standard 8051 microprocessor which can be operated at 48-MHz, 24-MHz, or 12-MHz
- 3.3V operation
- Smart Serial Interface Engine
- Vectored USB interrupts
- Separate data buffers for the SETUP and DATA portions of a CONTROL transfer
- Integrated I2C-compatible controller, runs at 100 or 400 kHz
- 32 general purpose I/Os

A.3.2 Block diagram

Appendix B

Schematic and PCB Layout

This appendix contains:

1. Front-end image sensor circuitry
2. EP1C6 FPGA Schematics
3. EZUSB FX2 Schematics
4. Top and Bottom PCB layout

Figure B.1: Image sensor board schematics

Figure B.2: EP1C6 FPGA Schematics

Figure B.3: FX2 Schematics

Figure B.4: Top-layer PCB layout

Figure B.5: Bottom-layer PCB layout

Appendix C

Command reference

C.1 FPGA command reference table

Appendix D

USB Basics

Peripheral architecture USB host uses a master/slave protocol to communicate with attached USB devices. It polls all connected peripherals at regular intervals of one milli-second. Each peripheral responds by placing its data on the time multiplexed bus at their allocated time within this 1milli-second frame. USB 1.x supports two speed modes low speed (1.5Mbits/sec) and full speed (12Mbit/sec). USB 2.0 supports high speed (480Mbits/sec) in addition to low speed and full speed.

In theory, a USB interface can support up to 127 individual USB peripherals at one time. The practical maximum number of devices is less since some of them reserve USB bandwidth. Figure D.1 depicts an example of USB peripheral topology where many devices are connected to a common bus via USB-hub.

USB uses a four-wire cable system. Two leads are used for power (5volt and gnd), the other two differential data lines. A device can be self powered, bus powered or both. Bus powered devices can draw up to 500mA current at 5V.

Figure D.1: USB peripheral devices topology

Figure D.2: USB descriptor format

D.1 Enumeration process and USB descriptors

When a USB device is attached to the bus it will be enumerated by the USB subsystem and then the host PC will learn all the device features by reading the descriptor. The descriptor is a data structure which contains information about the device and its properties. The USB standard defines a hierarchy of descriptors shown in figure D.2

Standard Descriptors :

- A Device Descriptor describes general information about a USB device. It includes information that applies globally to the device and all of the device s configurations. A USB device has only one device descriptor.

- The Configuration Descriptor gives information about a specific device configuration. A USB device has one or more configuration descriptors. Each configuration has one or more interfaces and each interface has zero or more endpoints. An endpoint is not shared among different interfaces within a single configuration, although a single interface can have several alternate settings which may use the same endpoint. Endpoints may be shared among interfaces that are part of different configurations without this restriction. Configurations can only be activated by the standard control transfer set_configuration. Different configurations can be used to change global device settings, such as power consumption.

- An Interface Descriptor describes a specific interface within a configuration. A configuration provides one or more interfaces, each with zero or more endpoint descriptors describing a unique set of endpoints within the configuration. An interface may include alternate settings that allow the endpoints and/or their characteristics to be varied after the device has been configured. The default setting for an interface is always alternate setting

zero. Alternate settings can be selected exclusively by the standard control transfer set_interface. For example a multifunctional device like a video camera with internal microphone could have three alternate settings to change the bandwidth allocation on the bus.

- An Endpoint Descriptor contains information required by the host to determine the bandwidth requirements of each endpoint. An endpoint represents a logical data source or sink of a USB device. The endpoint zero is used for all control transfers and there is never a descriptor for this endpoint. The USB specification uses the terms pipe and endpoint interchangeably.

- String Descriptors are optional and provide additional information in human readable unicode format. They can be used for vendor and device names or serial numbers.

D.2 Device classes

Standard device and interface descriptors contain fields that are related to classification: class, sub-class and protocol. These fields may be used by a host system to associate a device or interface to a driver, depending on how they are specified by the class specification.

Grouping devices or interfaces together in classes and then specifying the characteristics in a Class Specification offers several advantages. It makes device driver development easier and allows the development of host software which can manage multiple implementations based on that class. Such host software adapts its operation to a specific device or interface using descriptive information presented by the device. A class specification serves as a framework defining the minimum operation of all devices or interfaces which identify themselves as members of the class.

D.3 USB transaction

Most of USB transaction consists of a token, data and handshake packets. A token packet consist of a IN, OUT, or SETUP PID with address and EndPoint

Figure D.3: USB transaction

fields. For SETUP and OUT transactions, both the address and EndPoint fields identify the receiver of the data. For IN transactions, the address and EndPoint identify the sender of data to the host.

Data packets follow after the issue of one of the Token packets and depending on the type of transaction they may originate from either the host or the device. Handshake Generation. The final part of any transaction, except for isochronous transactions, is the handshake. The handshake consists of a PID sent by the receiver of the previous data, informing the sender of whether the data was successfully received, or if the data was received but need to be resent, whether a stall condition is present, or if the data was not received or received with errors.

D.4 USB transfer types

USB supports four transfer types : Control transfer, bulk transfer, interrupt transfer, and isochronous transfer.

Control transfers are used mainly to send control signals to the peripheral. These have high priority and incorporate inbuilt error protection. It is mainly used for transferring initialisation information, but can also be used for general-purpose low speed data transfers. All USB devices must support Control Transfers.

Bulk transfer is intended for transfers where a large amounts of data has to be transported in a time independent manner. This transfer type is typically used by mass storage devices, and printers. This method has low priority on the bus and hence it can wait if the bus is very busy.

Interrupt transfer is used by peripherals devices which must be polled at a regular interval. This transfer type is used for low speed devices such as mice and keyboards that need to send small amounts of data quickly and periodically to the PC.

Isochronous transfer is used in situations where data should be transferred to/ from the device in real-time at a constant data rate. In this transfer type data delivery is guaranteed however, no error correction is done. Isochronous transfer is used with streaming devices such as audio and video devices where data is produced at constant rate, and delay is not tolerated. Table D.1 gives full summary of the four transfer types.

Table D.1: USB transfer types summary

Appendix E

Device Driver

E.1 Operating and device driver interaction

E.2 Stream class and mini-driver interaction

E.3 SRBs processing by camera minidriver and USBCAMD

Appendix F

CD Appendix

The CD appendix contains the following materials:

1. Full device data sheets of the IBIS5-1300c image sensor, Cyclone FPGA family, Cypress EZUSB-FX2 (CY7C68013) and Voltage regulators (Max1692 and LT1763).
2. Complete VHDL code and Testbench files used in simulation
3. Matlab script files
4. Firmware code
5. Modified USB camera driver, installation package including INF file
6. Graphical user interface program (Developed in Visual C++) and
7. Video and still image results